POLITECNICO DI TORINO

Facoltà di Ingegneria
Corso di Laurea in Ingegneria Elettronica

Tesi di Laurea

Strumenti per la modellizzazione dei processi software

Relatori:
prof. Silvano Gai
dott. Maria Letizia Jaccheri

Candidato:
Mario Baldi

Maggio 1993

Ringraziamenti

Ringrazio i miei genitori per avermi saggiamente consigliato e sempre sostenuto durate gli studi.

Un ringraziamento è rivolto anche a Tomaso, Patricia, Giovanni, Ciro, Rosa, Massimo, Carmela ed Edoardo per la loro disponibilità ed il loro appoggio.

Indice

1

Introduzione

L'uso dei calcolatori elettronici assume importanza sempre maggiore ed il loro campo di applicazione si allarga continuamente. Questo corrisponde ad un continuo miglioramento delle prestazioni ed abbassamento dei prezzi. Per quanto riguarda l'hardware, le innovazioni tecnologiche e le esperienze accumulate nel settore, permettono di avere un andamento del rapporto prezzo/prestazioni in forte discesa. Lo stesso non si può dire riguardo al software, la cui produzione è resa più difficile da una notevole quantità di problemi.

Negli ultimi anni si sono fatti notevoli sforzi nell'intento di rendere più semplice, affidabile e soprattutto meno costosa, la produzione del software. In questo senso ci si è mossi in varie direzioni con due obiettivi principali:

1. definire processi per la produzione del software, analoghi a quelli per qualsiasi altra produzione industriale, che garantiscano un prodotto di qualità, ad un costo il più possibile basso e con l'utilizzo della minor quantità di risorse possibile;

2. fornire strumenti sempre più potenti ed efficienti per assistere le persone impegnate nelle varie fasi della produzione.

Questi due obiettivi sono strettamente connessi perché i processi di produzione sono tanto più efficienti, quanto più potenti sono gli strumenti che si hanno a disposizione per eseguirli.

1.1 Ambito di ricerca

Il lavoro presentato in questa trattazione è stato svolto in una ambito legato al primo dei due obiettivi sopraelencati, cioè la *modellizzazione dei processi software*, nota in letteratura come *Process Modeling* (PM). Questo filone di ricerca si occupa della definizione di processi che guidino la produzione del software, e a questo fine si possono identificare alcuni passi principali.

- Individuare i problemi e le carenze delle attuali metodologie per la produzione del software.
- Stabilire formalismi per la descrizione di processi software che permettano di:
  - definire rigorosamente tali processi
  - effettuare verifiche e simulazioni sulle descrizioni ottenute
  - eseguire i processi
  - modificare la definizione dei processi per aumentarne l'efficienza.
- Realizzare ambienti che permettano di eseguire e far evolvere i processi.
- Definire processi software efficienti e realmente utilizzabili per la produzione.

Al Dipartimento di Automatica e Informatica del Politecnico di Torino è iniziato, nel luglio 1992, un progetto per la realizzazione di E^3 (*Environment for Experimenting and Evolving process models*), un ambiente per l'esecuzione ed evoluzione dei modelli di processi software. Questo progetto si trova per ora in una fase iniziale ed è stato prodotto, nel febbraio 1993, il documento dei requisiti di E^3. Per la realizzazione di questo ambiente si è deciso di utilizzare tecniche proprie dell'orientamento agli oggetti (*Object Orientation*), paradigma che sta acquistando sempre maggiore credito e diffusione nell'ambito della prodizione del software.

1.2 Obiettivi del lavoro

Nel progetto E^3 si vuole utilizzare l'orientamento agli oggetti, non solo per la progettazione e realizzazione dell'ambiente, ma anche per la modellizzazione

dei processi. È allora necessario innanzitutto introdurre una possibile tecnica di modellizzazione ad oggetti dei procesi software, e quindi valutarne l'efficienza. Questo è stato l'obiettivo di fondo del lavoro che viene presentato in questa trattazione.

Per poter valutare la tecnica di modellizazzione è necessario avere a disposizione un ambiente PM che permetta di eseguire, o per lo meno simulare, i modelli di processo realizzati. A tale scopo si è deciso di realizzare S^3 (*Smalltalk environment for Software process Simulation*), un ambiente che fornisce supporto alla realizzazione ed alla simulazione di processi software secondo la tecnica di modellizzazione ad oggetti proposta. Le simulazioni devono essere realizzate con un elevato grado di concorrenza, ma non in un ambiente distribuito che sarebbe invece indispensabile per l'attuazione di processi. Le interfacce per la comunicazione tra le persone coinvolte nel processo ed il sistema, sono quindi eseguite sulla stessa macchina; così chi realizza la simulazione può tenere sotto controllo lo sviluppo del processo. Inoltre gli strumenti automatici che danno supporto agli utenti nelle varie attività che compongono il processo, non sono realmente eseguiti, ma la loro esecuzione è semplicemente simulata.

Anche per la realizzazione di S^3 si vuole utilizzare l'orientamento agli oggetti; per la progettazione dell'ambiente, così come per quella dei modelli, si vuole utilizzare la metodologia Coad/Yourdon sfruttando il supporto offerto da DECdesign. L'utilizzo di questo strumento permette di valutarne l'efficienza per decidere se possa essere vantaggioso usarlo anche come strumento fornito da E^3 per la realizzazione dei modelli di processo.

1.3 Fasi del lavoro

Il lavoro svolto si è articolato in tre fasi principali.

1. Una ricerca in letteratura con lo scopo di definire lo stato dell'arte nella modellizzazione dei processi software.

   Questa fase ha portato anche alla stesura di un elenco di definizioni che si possono considerare alla base della modellizzazione dei processi, e che sono state incluse nel documento dei requisiti di E^3.

2. Progettazione e realizzazione di S^3.

La progettazione è stata fatta utilizzando la metodologia ad oggetti proposta da Coad e Yourdon [CY91a] [CY91b] e servendosi di DECdesign [D.E92b], uno strumento automatico di supporto alla progettazione.

La realizzazione è stata fatta utilizzando Objectworks(r)/Smalltalk [Sys90], un linguaggio di programmazione orientato agli oggetti. S^3 è un insieme di classi che forniscono le funzionalità tipiche dell'ambiente di simulazione e che cosituiscono uno schema predefinito su cui fondare la modellizzazione dei processi da simulare. Quindi a S^3 è associata una tecnica di modellizzazione ad oggetti che si basa sui meccanismi e gli strumenti che S^3 stesso mette a disposizione.

3. Modelizzazione di un processo secondo la tecnica supportata da S^3 e simulazione.

   Definito un semplice processo esemplificativo, se ne è fatto un modello ad oggetti utilizzando la metodologia Coad/Yourdon e lo si è implementato con gli strumenti messi a disposizione da S^3. Questo esempio di processo è stato simulato permettendo così di valutare l'efficienza di S^3.

1.4 Struttura della trattazione

La trattazione che riporta il lavoro fatto per raggiungere gli obiettivi esposti nelle sezioni precedenti, è strutturata in quattro parti seguite da una serie di appendici.

Modellizzazione dei processi software

La prima parte contiene le informazioni ricavate studiando quanto è riportato in letteratura riguardo alla modellizzazione dei processi software, in modo da conoscere i problemi principali che caratterizzano questo ambito e le principali soluzioni adottate.

Nel capitolo 2 sono riportati i concetti alla base della modellizzazione dei processi ed alcune considerazioni sui vantaggi a cui il PM porta. Infine sono riportate le caratteristiche che un linguaggio per la modellizzazione dei processi deve possedere.

Si è detto che si vuole valutare l'efficacia del paradigma ad oggetti applicandolo alla modellizzazione dei processi; in questa prospettiva, il modello del processo, per lo meno nella sua versione eseguibile, sarà una descrizione in un linguaggio di programmazione ad oggetti. Per questo si è posta particolare enfasi sugli approcci ed ambienti PM che realizzano i modelli mediante un linguaggio di programmazione. Il capitolo 3 riporta appunto uno studio sulla *programmazione di processo* con riferimento ai principali ambienti PM noti in letteratura.

Nel capitolo 4 è riportato un estratto delle informazioni raccolte in letteratura riguardo agli altri approcci di modellizzazione dei processi ed i rispettivi ambienti.

L'orientamento agli oggetti

Nel capitolo 5 sono riportate le principali caratteristiche ed i punti di forza dell'orientamento agli oggetti, soprattutto per quanto riguarda l'analisi e la progettazione ad oggetti. Infatti queste si basano su principi molto generali, la maggior parte dei quali sono anche alla base della programmazione ad oggetti.

Nel capitolo 6 è trattata nel dattaglio la metodologia di progettazione ad oggetti Coad/Yourdon che sarà utilizzata per realizzare il progetto di S^3 ed anche per la modellizzazione dell'esempio di processo proposto.

Si è deciso di studiare la metodologia di progettazione proposta da Booch perché è molto potente e dettagliata, anche se non è utilizzata direttamente in questo lavoro. Essa fornisce un ottimo termine di confronto per apprezzare vantaggi e svantaggi della metodologia Coad/Yourdon. Una breve descrizione degli aspetti principali della metodologia Booch è riortata nel capitolo 7.

Progetto di S^3

Nel capitolo 8 sono trattati gli aspetti generali dell'ambiente di simulazione ed i principi su cui si basa il suo funzionamento. Particolare enfasi è posta sulla gestione delle attività che costituisce l'aspetto principale dell'esecuzione dei processi software.

Le classi che costituiscono il progetto di S^3 sono presentate nel capitolo 9 descrivendo le funzionalità che esse mettono a disposizione.

Nel capitolo 10 vengono esposti i meccanismi su cui si basa il funzionamento di S^3 ed in particolare la simulazione dei processi. In queste tematiche rientra la spiegazione del ciclo di vita delle attività, descrivendo i vari stati in cui queste vengono a trovarsi e gli eventi che gestiscono in ognuno di questi stati. Infine sono fatte alcune considerazioni su come il progetto si adatta all'uso della concorrenza e della distribuzione.

La progettazione di S^3 e la definizione dello schema di classi predefinite che mette a disposizione per la modellizzazione dei processi, è accompagnata dalla definizione di una tecnica ad oggetti per la modellizzazione dei processi e da un relativo processo (detto meta-processo) per la creazione ed esecuzione dei modelli di processo. Entrambe questi argomenti sono trattati nel capitolo 11 dove è riportata anche la definizione di un esempio di processo software sul quale provare la bontà della tecnica di modellizzazione proposta. Quindi sono state riportate le parti principlali del modello dell'esempio di processo, ponendo particolare enfasi su come siano tradotti nel modello alcuni requisiti richiesti dalle specifiche del processo. Infine è riportato un esempio di potenziamento di uno dei meccanismi offerti da S^3 ed un frammento di modello in cui sono applicati questi meccanismi potenziati.

Implementazione di S^3

Nel capitolo 12 sono riportati alcuni aspetti generali dell'implementazione di S^3, cioè la struttura dell'insieme di classi che costituiscono l'ambiente. Sono espresse una serie di considerazioni sull'utilizzo della concorrenza nella simulazione dei processi e sono descritte alcune delle classi di utilità di S^3.

La definizione delle classi introdotte nel progetto è descritta nel capitolo 13 in cui sono spiegati nel dettaglio i principali meccanismi che caratterizzano S^3.

Nel capitolo 14 sono riportati gli obiettivi raggiunti e le valutazioni che il lavoro ha permesso di fare in prospettiva alla realizzazione di E^3. Infine sono individuati alcuni possibili sviluppi futuri.

Appendici

Nell'appendice A sono riportate le caratteristiche principali di DECdesign che è utilizzato per la progettazione di S^3 e per la realizzazione dei modelli di processo secondo la tecnica proposta. In questa appendice sono descritte

le modalità in cui è dato supporto per la metodologia Coad/Yourdon e i vantaggi e gli svantaggi che derivano dall'utilizzo di DECdesign.

Nell'appendice B si è data una visione di insieme di Smalltalk-80 e dell'ambiente di programmazione offerto da Objectworks(r)/Smalltalk. Sono presentate alcune delle più importanti classi predefinite che sono state utili nell'implementazione di S^3.

Nell'appendice C sono state riportate le definizioni di tutte le classi che compongono S^3, raggruppandole nelle categorie in cui sono state poste all'interno di Objectworks(r)/Smalltalk. S^3 è composto da più di 4200 linee di codice Smalltalk-80.

L'appendice D contiene l'articolo *Object Oriented Process Modeling* scritto dall'autore di questa trattazione e dai due relatori e sottomesso alla conferenza TOOLS (Technology of Object-Oriented Languages and Systems) Santa Barbara, California, 2-5 agosto 1993. L'articolo riporta il lavoro esposto in questa trattazione ed i risultati ottenuti, mettendo particolarmente in rilievo gli aspetti legati all'utilizzo di metodologie ad oggetti per la realizzazione sia dell'ambiente, sia dei modelli di processo.

Parte I

Modellizzazione dei processi software

2

Modelli di processo

In questo capitolo sono trattati alcuni concetti fondamentali riguardanti i processi software e la loro modellizzazione. Sono quindi riportate alcune delle definizioni che si sono adottate nell'ambito del progetto E^3 e che coincidono quasi completamente con quelle più consolidate che si possono trovare in letteratura. Infine sono fatte alcune considerazioni sui vantaggi derivanti dalla modellizzazione dei processi e sulle caratteristiche che devono avere i linguaggi di modellizzazione per poter facilmente ottenere tali vantaggi.

2.1 Processi e processi software

Un *processo* è un appbroccio sistematico per creare un prodotto o per portare a termine un compito [Ost87].

Meno in generale, un processo di produzione serve a produrre prodotti di qualità in modo affidabile, efficiente e con la possibilità di stabilire, o per lo meno stimare, a priori i tempi ed i budget necessari. Il software viene prodotto utilizzando processi più o meno formali che si vogliono studiare per meglio capirli, formalizzarli, eseguirli in modo il più possibile automatico e migliorarli.

Il processo per la produzione del software, detto concisamente *processo software*[1], deve coprire tutte le diverse fasi della produzione di programmi, dal concepimento dell'idea con conseguente definizione delle specifiche, alla consegna del prodotto e sua evoluzione.

La descrizione di un processo è l'insieme delle azioni che devono essere

[1]Il termine software, quando non ambiguo, sarà talvolta omesso.

eseguite per ottenere il prodotto voluto. Tale descrizione spesso dà anche l'ordine con cui queste azioni devono essere intraprese. Un *modello del processo* (per la produzione del software) è una descrizione di una classe di processi software.

Analogamente si può definire un processo per produrre modelli di processo e questo sarà detto *meta-processo software*; la sua descrizione sarà quindi il *modello del meta-processo software*.

Lo sviluppo del software, come di ogni altro prodotto, può quindi essere fatto seguendo un certo modello di processo. I processi per la produzione del software hanno caratteristiche peculiari rispetto agli altri processi di produzione perché il prodotto dei processi software è qualcosa di invisibile e non tangibile. Inoltre nella descrizione di processi per la produzione del software non si è aiutati dal fatto di dover sottostare a regole di carattere fisico o chimico, come accade nel caso dei normali processi produttivi.

Si è notato che, in svariati campi, pur non descrivendo i processi in modo dettagliato e preciso, l'uomo riesce comunque ad usarli in maniera efficiente e produttiva. Ciò è dovuto all'innata propensione che ha l'uomo nell'uso di metodologie che prevedono descrizione, istanziazione ed esecuzione di un processo.

Una ulteriore differenza tra processi software e normali processi di produzione, sta nel fatto che il grado di complicazione dei processi software è difficilmente predicibile a priori ed essi si rivelano sempre più complicati di quanto ci si aspetti.

Un gran numero di persone è coinvolto nell'attuazione di un processo software; questo fatto richiede di dedicare una particolare attenzione al miglioramento della comunicazione tra gli individui coinvolti nel progetto.

La descrizione dei processi è importante anche perché rappresenta un mezzo di comunicazione tra il responsabile della produzione di un certo prodotto e coloro che lavorano al fine di realizzare tale produzione. Infatti grazie a questa descrizione il responsabile della produzione può comunicare ad ogni membro del gruppo del progetto quali sono le azioni che deve intraprendere per portare avanti lo sviluppo del prodotto ed anche quale deve essere l'ordine ed il momento in cui le varie azioni vanno fatte.

Inoltre la descrizione del processo permette la comunicazione tra responsabili della produzione di diversi prodotti che possono così condividere le loro conoscenze. Soprattutto, un aspetto non trascurabile, è la possibilità di

riutilizzare procedimenti ed approcci seguiti da altre persone, o anche dalla stessa, in altri progetti. Ciò permette, tra l'altro, la trasmissione di una conoscenza che diversamente resterebbe confinata nell'esperienza del singolo individuo.

2.2 Modelli di processi software e processi software

Non esiste un processo software valido per la produzione di qualunque tipo di programma, ma il processo deve essere adattato ad ogni singola produzione. In particolare esso va adattato al tipo di prodotto, alle necessità e preferenze del responsabile del progetto ed anche alla struttura dell'oganizzazione ed alle risorse della casa di produzione.

Neppure un "linguaggio" per la descrizione del processo è universalmente affermato, anche perché la sua efficacia dipende dell'ambiente in cui è usato e dagli obiettivi che si vogliono raggiungere con esso [Rom90].

Come si è data una definizione generale del concetto di processo, si può dare una definizione generale del concetto di *processo software*. Questo è l'insieme delle attività di ingegneria del software necessarie per trasformare le specifiche dell'utente in un prodotto funzionante e per gestirne l'evoluzione [Jac91].

In generale si può creare un *modello di processo software generico* che fornisce definizioni, relazioni e strutture dei vari elementi di un processo, così che è possibile stabilire tecnologie e metodi applicabili a differenti tipi di progetto. Proprio per questa sua vasta applicabilità questo modello fornisce una descrizione molto ad alto livello che in genere non fa altro che definire l'ordine tra passi della produzione quali analisi, progetto, codifica, ecc.

Quindi si può specificare un *modello di processo software specifico del progetto*; questo è una personalizzazione del modello di processo software generico adattata alle necessità di un particolare progetto governato da una metodologia specifica e diretto allo sviluppo di un ben preciso tipo di software. Inoltre il processo è adattato anche alle disponibilità di risorse e all'organizzazione della casa produttrice, nonché alle preferenze del responsabile del processo.

Viene quindi creata un'istanza del modello del processo software specifico del progetto (o semplicemente modello di processo) che è attuabile (verrà

dettagliatamente spiegato in seguito il concetto di attuazione o '*enactment*'); essa consiste di un insieme di attività connesse ai prodotti che hanno in ingresso e in uscita, agli strumenti di cui necessitano e ai ruoli responsabili del loro sviluppo cui sono state associate le persone [JC93]. Tutto ciò è stato fatto per adattare il modello del processo allo sviluppo di un particolare prodotto. L'attuazione di questa istanza del modello di processo è il *processo software*.

Va tenuto presente che i gradi di dettaglio e di specificità che devono raggiungere i vari tipi di modello non sono chiaramente ed universalmente stabiliti, ma dipendono da scelte proprie della filosofia di produzione che si vuole adottare.

Un aspetto non trascurabile è che non solo il prodotto del processo è soggetto a continue evoluzioni e modifiche, ma anche il modello stesso del processo. Questo, come si vedrà in seguito, riveste una grande importanza ed ha come conseguenza una notevole difficoltà implementativa soprattutto perché queste modifiche devono essere possibili durante l'attuazione del processo e senza richiedere di ricominciarne l'attuazione dal principio.

Anche la produzione di un modello di processo può essere vista come un processo, o meglio un *meta-processo* che guida l'analisi dei requisiti, il progetto, la personalizzazione, l'istanziazione, l'attuazione e l'evoluzione di un modello di processo. I passi sopraelencati definiscono un esempio di *ciclo di vita del processo*; la definizione dei passi e delle loro competenze sono fortemente dipendenti dalle scelte di una certa filosofia di modellizzazione dei processi che va fatta a monte di tutte le scelte riguardanti la produzione.

2.3 Definizioni e terminologia

In questa sezione sono riportate le definizioni dei principali termini utilizzati nella modellizzazione dei processi che sono state ricavate per lo più da quelle date in [Lon92]. Per ogni termine è dato anche l'equivamente inglese.

Processo software ('software process') È l'insieme di tutte le entità e le attività del mondo reale che sono coinvolte nello sviluppo di un sistema software.

Attività ('task') Rappresenta un passo del *processo*. Essa può produrre qualcosa o anche semplicemente avere funzioni di coordinamento all'interno del *processo*.

Risorsa (‘resource’) Indica un bene messo a disposizione di un’*attività* che ne ha bisogno per essere eseguita. Questi beni possono essere di diversa natura: tempo, strumenti automatici, macchine, ecc.

Prodotto (‘artifact’) Un entità creata o modificata da un’*attività* durante un *processo*, o perché è un risultato richiesto, oppure perché è utile a facilitare il *processo*.

Modello di processo software (‘software process model’) Descrizione di una classe di *processi software*.

Modello del ciclo di vita (‘life cycle model’) *Modello di processo software* informale e poco dettagliato.

Ambiente per la modellizzazione dei processi (‘PM environment’) Fornisce adeguato supporto e strumenti automatici per la *modellizzazione di processi software*.

Linguaggio di modellizzazione dei processi (‘PM language’) Un formalismo o un linguaggio in grado di rappresentare *modelli di processi software*.

Meta-processo (‘meta-process’) È l’insieme di tutte le entità e le attività del mondo reale che sono coinvolte nello sviluppo di un modello di processo software.

Modello del meta-processo (‘meta-process model’) Descrizione di una classe di *meta-processi*.

Meta-modello del processo (‘process meta-model’) È una struttura che consente di rappresentare ed esprimere i *modelli di processo* di cui fa parte anche il *modello del meta-processo*. Non si tratta di strumenti da utilizzare per realizzare il modello, ma per descriverlo.

Ruolo (‘role’) Un insieme capacità e permessi; ogni persona partecipa al processo ricoprendo un ruolo.

Esecutore del modello di processo (‘process model performer’) È coinvolto nel *processo software* come esecutore di alcune attività che lo compongono.

Ingegnere di modelli di processo ('process model engineer') Ha il compito di produrre il *modello del processo*. Quindi è un esecutore del meta-processo.

Responsabile del progetto ('project manager') Il suo compito è di fornire informazioni per istanziare il *modello del processo* e di eseguire alcune attività dell'attuazione di questo. Questo è un ruolo che ha un particolare rilievo ed è coinvolto in qualunque *processo software*.

2.4 Vantaggi della modellizzazione dei processi

In questa sezione sono descritti i principali vantaggi individuati in letteratura che derivano dalla modellizzazione dei processi software. Da questi derivano gli obiettivi che ci si deve porre nel realizzare i modelli e gli ambienti che ne permettono l'esecuzione.

In particolare alcuni di questi sono più strettamente legati al *modello* del processo ed altri all'*ambiente* di modellizzazione ed esecuzione dei processi.

2.4.1 Obiettivi del modello di processo

Comunicazione

Come detto in precedenza uno dei vantaggi nell'utilizzo di modelli di processo sta nella possibilità di fornire un efficiente mezzo di *comunicazione* tra le persone che producono e gestiscono l'evoluzione di un prodotto software. Infatti il modello del processo aiuta a dare a tutti una adeguata e comune conoscenza del problema.

Previsione

Un altro obiettivo molto importante da conseguire grazie alla modellizzazione dei processi è quello di dare uno strumento con il quale *stimare* le risorse ed i tempi necessari per portare a termine il processo in questione. Secondo le metodologie attuali questo incarico è affidato al responsabile del processo, ma il modello del processo costituisce un buon mezzo di supporto su cui basarsi per fare queste stime in modo più semplice e per confermarle o meno.

Inoltre grazie alla formalizzazione di un modello del processo è possibile fare delle simulazioni per vedere l'effetto di decisioni possibili, in modo da poter scegliere quella migliore nel modo più oculato possibile. Oppure ancora, grazie alle archiviazioni fatte in precedenza, si possono considerare circostanze simili a quelle in esame per vedere come si è proceduto, magari anche da parte di altre persone, per trarne dei suggerimenti.

Completezza

Perché questo possa essere fatto in modo efficiente ed utile è necessario che il modello *copra* tutte le *fasi* del ciclo di vita di un prodotto software che si estende dalla definizione dei requisiti che dovrà avere il prodotto finale, alla progettazione, all'implementazione, la verifica, l'installazione, alla creazione della documentazione ed infine al mantenimento del prodotto e della relativa documentazione.

Inoltre il modello deve considerare anche tutte le *entità* coinvolte nel processo, cioè i prodotti software, le attività necessarie per l'attuazione del processo–siano esse di tipo automatico o fatte da persone–, gli strumenti software, le risorse hardware e le persone stesse con i loro ruoli. Nella modellizzazione delle attività, soprattutto quelle fatte da persone, è indispensabile tenere conto anche delle *tempistiche* per la loro attuazione per poi ottenere simulazioni significative ed utili e poter fare pervisioni fondate.

Nella modellizzazione delle persone si deve considerare anche il *carico di lavoro* cui sono sottoposti. Infine il modello deve tener conto della *limitatezza* di tutte quante le *risorse* che intervengono nel processo.

Adattabilità

Come già detto in precedenza non è possibile definire un modello di processo utilizzabile per qualsiasi prodotto. Inoltre, dato un modello di processo per una certa produzione, questo va adattato alle disponibilità di risorse ed alle preferenze del responsabile del progetto. Quindi è necessario rendere possibile una *configurazione* del modello di processo prima della sua istanziazione. Ciò va fatto fin dall'implementazione del modello prevedendo soluzioni alternative o parametrizzabili. Nel secondo caso la differenza tra modelli sarà effettiva solo al momento dell'istanziazione.

Questo aspetto richiede quindi di introdurre modifiche al modello prima di eseguirlo (non si tratta di evoluzione trattata nel seguito) e adattarlo a certe

specifiche caratteristiche di un progetto. Tutta la problematica che nasce da questo è detta *personalizzazione* ('*customization*') del modello del processo e richiede di affrontare problemi diversi rispetto a quelli della evoluzione; proprio per questo bisogna avere ben presente la distinzione tra le due.

Inoltre quando il modello del processo evolve, devono evolvere anche le diverse configurazioni che sono state generate per necessità di personalizzazione, così che nasce tutta una problematica legata alla gestione delle configurazioni e loro evoluzione ('*Configuration Management*').

Questa tematica è presente anche nei prodotti software evidenziando ancora una volta il fatto che i modelli di processo sono anch'essi software. Allora, come espresso in [Jac91] o più esplicitamente in [JC93], la gestione delle configurazioni per un certo prodotto è guidata dal modello del processo, però nello stesso tempo i modelli di processo devono essere trattati con i metodi propri della gestione delle configurazioni.

Evoluzione

Un ambiente per l'attuazione di modelli di processo deve permettere al responsabile di processo di *controllare* che le varie fasi dell'esecuzione del processo vadano a buon fine avendo anche uno strumento per comprendere se le scelte di pianificazione e di suddivisione del lavoro siano state giuste.

Perché ciò porti dei vantaggi deve esserci la possibilità da parte del responsabile di processo di *modificare* il modello del processo, anche mentre questo è in esecuzione. I cambiamenti fatti però possono portare ad uno stato di inconsistenza che va gestito dall'ambiente di attuazione del processo riportando gradualmente la situazione ad uno stato di consistenza.

Questi cambiamenti costituiscono la così detta *evoluzione* ('*evolution*') del modello del processo e possono essere dovuti a diversi fattori tra cui [Mad91]:

- modello di processo non ben funzionante.
- aggiunta di nuove specifiche ('requirments') per il modello di processo, modifica o eliminazione di altre non più valide.
- acquisizione di una conoscenza più profonda durante lo sviluppo ed esecuzione del processo.
- prestazioni (tempi, qualità, costi) non soddisfacenti richiedono una modifica anche per mantenere competitività sul mercato.

- necessità di personalizzare un modello di processo troppo generico per ottenere alcuni risultati specifici.
- gli sviluppi della tecnologia o i cambiamenti delle preferenze delle persone che lavorano al progetto.
- deviazione dai piani previsti per l'uso delle risorse e la distribuzione dei carichi di lavoro.
- cambiamento delle assegnazioni dei diversi compiti ('tasks') alle varie persone.
- variazione delle precedenze tra le varie attività. Questo è l'aspetto che riveste maggiore importanza.

Va tenuto presente che i cambiamenti non sono fatti solamente perché, in seguito a valutazioni e misure, ci si rende conto di possibili migliorie, ma sono intrinseci al fatto che all'attuazione del modello del processo cooperano delle persone. Infatti questi tendono a modificare e migliorare (almeno negli intenti) il processo che stanno portando avanti perché lavorandoci sopra acquisiscono una più profonda conoscenza del problema e degli obiettivi da raggiungere.

Dietro a questo c'è anche un motivo psicologico, cioè la necessità di sentire che stanno conducendo una attività creativa e non puramente meccanica.

I cambiamenti fatti ai modelli di processo costituiscono un problema non banale, non solamente perché devono essere possibili durante l'attuazione del processo stesso, ma anche perché si devono essere estesi alle varie versioni del modello in modo da gestire coerentemente la sua evoluzione. Così si vede nuovamente come il modello del processo richieda gli stessi trattamenti del prodotto della sua esecuzione, cioè il software ("I modelli di processi software sono anch'essi software" [Ost87]).

Riutilizzo

Un vantaggio della modellizzazione dei processi sta nel fatto che non è il caso di definire ogni volta un modello a partire dal nulla, ma si possono *riutilizzare* modelli, o parti di essi, già definiti e dimostratisi efficaci. Ciò si collega alla affermazione del fatto che i modelli di processi permettono di fissare e trasmettere conoscenze che diversamente sarebbero confinate nella mente di pochi addetti ai lavori.

Se non si ricorresse ai modelli di processi, l'unica forma di riutilizzo delle esperienze fatte lavorando ad alcuni progetti starebbe nell'assegnare a diversi progetti le persone detentrici di questa conoscenza.

2.4.2 Obiettivi di un ambiente PM

Verifica

La modellizzazione dei processi di produzione del software, se integrata in un ambiente efficiente, permette di svolgere attività di *verifica* della bontà del processo ad esempio mediante la simulazione dello stesso o mediante altri metodi dipendenti anche dal tipo di rappresentazione scelta per i modelli.

Valutazione

Importante vantaggio della modellizzazione dei processi deve essere la possibilità di effettuare *misure* e valutazioni, sia sul modello del processo che, durante la sua esecuzione, sul prodotto in via di sviluppo.

Questo richiede di inserire dei punti di misura nel modello del processo in corrispondenza dei quali sarà possibile, durante l'attuazione del processo, effettuare e raccogliere in modo più o meno automatico delle misure. Queste possono essere utili in ambito di valutazione per capire se siano necessarie modifiche o migliorie. Eventualmente questo può servire come bagaglio di esperienza per capire se sarebbe bene agire nello stesso modo quando in altri progetti ci si trovasse in circostanze simili. Inoltre la modellizzazione dei processi permette di confrontare differenti processi tra loro e capire quali siano i relativi vantaggie e svantaggi.

Attuazione

È importante che il modello del processo possa essere *attuato*. Sebbene, come detto più volte, sotto molti punti di vista il modello di processo si possa vedere come un qualunque prodotto software, per indicarne l'esecuzione in letteratura non si usa quasi mai la parola 'execution', come per i programmi, ma la parola 'enactment'. Questo sta ad evidenziare che un processo software non si limita ad una pura sequenza di operazioni fatte da calcolatori, ma che c'è una stretta interazione con le persone che sono gli unici che possono svolgere determinate parti del processo.

Il modello del processo può essere interpretato manualmente o automaticamente [Mad91].

Le descrizioni di processo *interpretate manualmente* guidano le persone indicando loro quali sono le attività necessarie a sviluppare un certo prodotto. Questi tipi di processi sono generalmente meno rigorosi e lasciano spazio all'operatore nel prendere decisioni in un certo ambito che non è noto a priori. Per questo l'interpretazione manuale è maggiormente utilizzata quando in un processo software ci siano incertezza e variabilità o quando siano richieste decisioni creative.

Questo tipo di attuazione si rivela particolarmente utile nelle attività di alto livello di tipo gestionale che hanno a che fare con la sensibilità o i rapporti umani, la soddisfazionepersonale, ecc. Oppure quando si hanno rappresentazioni non facilmente strutturabili o fortemente dipendenti dal contesto e specialistiche.

Altro campo di utilità è quello di attività di basso livello molto tecniche come progettazione o revisione che richiedono capacità di creatività e di giudizio e possono essere soggette ad eventi impredicibili.

Ci sono poi descrizioni di processi *interpretate dalle macchine* che sono utili per portare a compimento piani preprogrammati guidati in ciò dai calcolatori in grado di coordinare le varie fasi e di svolgerne direttamente alcune. All'interno di questi processi ci possono essere attività svolte da persone per le quali la macchina attende i risultati dell'attività umana.

Queste attuazioni automatiche sono maggiormente utilizzate per processi ormai consolidati, ben adattati ed in generale soggetti a pochi cambiamenti. Ovviamente è bene rivolgersi sempre più a questo tipo di processi piuttosto che a quelli interpretati manualmente.

La creazione del software non è una pura attività meccanica, ma è qualcosa di creativo; essa comprende sia attività di tipo tecnico, ad esempio la vera e propria codifica, che attività di gestone, come la pianificazione dell'uso delle risorse e la distribuzione dei carichi di lavoro. Tutte queste attività sono svolte in concorrenza da persone diverse e magari geograficamente piuttosto lontane.

Comunque vengano interpretati, si possono distinguere due differenti tipi di modelli di processo [Mad91]. I processi *prescrittivi* ('*prescriptive*') definiscono quali sono le attività da portare a compimento ed in quale ordine ciò va fatto. I processi *proscrittivi* ('*proscriptive*') non fanno altro che proibire azioni che non vanno fatte perché il processo venga fatto avanzare secondo

lo schema prescelto, invece che mostrare quelle da fare.

2.5 Caratteristiche di un linguaggio per la modellizzazione di processi

In questa trattazione si parla di linguaggio in senso lato perché un linguaggio per la modellizzazione dei processi non vuole essere necessariamente un linguaggio formale, ma si può intendere anche una rappresentazione di tipo grafico o di altro genere. Il problema della definizione di un mezzo efficiente per la modellizzazione dei processi software è stato affrontato seguendo diversi approcci, ma nessuno singolarmente ha fornito un prodotto efficiente e completo. Questo è dovuto principalmente al fatto che la modellizzazione dei processi è un problema differente e maggiormente complesso rispetto a quelli normalmente trattati nell'ambito della produzione del software.

Uno degli approcci più comuni è quello di utilizzare come linguaggio di specifica per modelli di processo un linguaggio esistente ed applicato ad altre aree, magari con qualche modifica. Infatti tra le caratteristiche che si possono evidenziare come necessarie per un linguaggio per la modellizzazione dei processi, ce ne sono alcune tipiche di linguaggi già esistenti nel campo della programmazione sequenziale, concorrente, per sistemi in tempo reale o per basi di dati. Queste caratteristiche, sebbene indispensabili per la definizione di un buon linguaggio, non sembrano di per sé sufficienti per dare un valido supporto alla modellizzazione dei processi.

2.5.1 Caratteristiche ereditabili da linguaggi esistenti

Meccanismi di astrazione Permettono di focalizzare l'attenzione su aspetti generali e più significativi trascurando i dettagli implementativi.

Modularità Dà la possibilità di strutturare il modello del processo in unità distinte e separate, anche dal punto di vista fisico, le quali comunque interagiscono tra loro. La modularità è presente anche nel prodotto stesso del processo, cioè nei programmi, che sono normalmente suddivisi in vari moduli. L'indipendenza dei moduli fornisce svariati vantaggi tra cui una maggior maneggevolezza, dal momento che questi possono avere dimensioni non troppo

grandi, e minori problemi per le modifiche, perché queste possono essere fatte separatamente e magari concorrentemente su moduli distinti. L'uso della modularità richiede la definizione di interfacce ben precise tra i vari moduli che devono avere interazioni, però i vantaggi provenienti dalla suddivisione compensano lo sforzo fatto in questo senso.

Genericità È bene avere la possibilità di realizzare soluzioni applicabili a casi differenti ad esempio mediante parametrizzazione o istanziazione.

Non determinismo Fornisce la possibilità di non esprimere a priori quale scelta sarà fatta in una situazione che permette più di una alternativa. Questa non è una proprietà molto diffusa tra i linguaggi di programmazione (se non nel Prolog) che tendono generalmente a dare l'ordine preciso in cui sono condotte le varie azioni.

Meccanismi per la definizione di tipi Consentono di creare ed utilizzare strutture di dati astratte che possono essere utili perché nella modellizzazione di processi si ha spesso a che fare con strutture dati molto complesse.

Concorrenza Come detto più volte, l'attività di sviluppo del software, e cioè l'esecuzione di un processo software, è intrinsecamente concorrente perché ci sono differenti strumenti automatici e numerose persone che agiscono contemporaneamente ed interagiscono tra loro. Il linguaggio utilizzato per modellizzare i processi deve supportare la concorrenza e mettere a disposizione meccanismi di sincronizzazione e di comunicazione.

In questo contesto si possono evidenziare tutta una serie di caratteristiche ereditabili dai linguaggi per sistemi in tempo reale o per sistemi reattivi. Infatti le attività che permettono l'esecuzione di un modello di processo vanno fatte in un ben determinato ordine ed entro certi tempi ben precisi. Inoltre alcune azioni possono essere intraprese solo dopo che determinate attività siano giunte a corretta conclusione o in risposta ad altro tipo di eventi attivati da strumenti automatici o da persone che lavorano nell'ambito del progetto. Una differenza con i sistemi in tempo reale sta nel fatto che i tempi sono di diversi ordini di grandezza superiori, ma i concetti alla base sono gli stessi.

Vincoli temporali Le varie attività vanno attivate in un ordine ben preciso dipendente, sia da eventi esterni, che da condizioni di precedenza. Poiché

la corretta esecuzione del modello del processo non dipende solamente dall'ordine delle varie attività e dalla loro corretta esecuzione, ma anche dal rispetto di ben precisi vincoli temporali, l'ambiente deve garantire che vengano rispettate delle tempistiche rigide (i così detti '*deadlines*').

Dunque è necessario che oltre a fornire strumenti per la gestione del parallelismo, si abbia la possibilità di stabilire e gestire *vincoli temporali* e l'*interazione con l'ambiente*.

Gestione dei dati

Dati di dimensioni molto diverse Sia nella modellizzazione che nell'attuazione di processi, è indispensabile memorizzare e recuperare da archivio dei dati, per cui sono necessarie per il linguaggio di definizione caratteristiche molto simili a quelle tipiche dei linguaggi che operano su basi di dati. Ci sono però aspetti peculiari della modellizzazione dei processi quali la capacità di maneggiare dati di dimensioni notevoli, come ad esempio interi listati di sorgente o moduli di codice.

A complicare il problema si pone il fatto che si rivela spesso necessario anche trattare con dati di dimensioni molto piccole come può essere un singolo nodo di un albero di analisi sintattica. Allora deve essere presente la capacità di *trattare dati sia di piccole che di grandi dimensioni*.

Transazioni lunghe Le transazioni sulle basi di dati devono poter essere molto lunghe ('*long transaction*') perché le elaborazioni dei dati contenuti (ad esempio modifica di sorgenti) possono durare anche dei giorni.

Transazioni cooperative A produrre un ulteriore grado di difficoltà sta il fatto che più persone o strumenti possono contemporaneamente richiedere l'accesso ad uno stesso dato ('*cooperative transaction*'). Però, proprio per la notevole durata delle transazioni, non è possibile utilizzare i classici metodi di bloccaggio e di ritorno ad uno stato precedentemente memorizzato in caso di errori o problemi ('rollback').

Versionamento Poiché sia il modello del processo che i prodotti evolvono, è necessario utilizzare basi di dati *versionate*, cioè in grado di gestire le diverse versioni con le varie modifiche dei dati memorizzati in esse.

Modificabilità del modello

Si è già evidenziato più volte il fatto che il modello del processo possa cambiare (come anche i prodotti software). Questo è legato principalmente a due aspetti:

1. *personalizzazione*, ovvero la necessità di adattare il modello del processo alle esigenze del suo responsabile, allo specifico pregetto ed alla casa produttrice che lo sviluppa;

2. *evoluzione*: come evolve il prodotto, così evolve anche il modello del processo perché, anche durante la sua attuazione, si rivelano necessari cambiamenti che, se trascurati, renderebbero inefficiente il processo.

Per gestire questi cambiamenti ci sono diversi meccanismi oltre a quello del versionamento trattato in precedenza. Questi meccanismi impongono al linguaggio di modellizzazione ben determinate caratteristiche.

Sottotipi È vantaggioso che il linguaggio di modellizzazione fornisca la possibilità di definire diversi *sottotipi* di un certo tipo, permettendo di aggiungere alcune caratteristiche a quelle del tipo padre. Creando numerosi sottotipi si ottengono versioni rifinite e personalizzate del tipo padre da cui questi discendono. Questo meccanismo è paricolarmente adatto per la gestione della personalizzazione del modello del processo piuttosto che della sua evoluzione.

Tipi laterali Si tratta di un meccanismo molto simile a quello dei sottotipi, ma invece che realizzare una personalizzazione aggiungendo nuove caratteristiche ad un tipo di dato astratto, si crea un tipo di dato differente. Quest'ultimo, oltre ad avere alcune caratteristiche in più, può anche non averne alcune. Così tra i due tipi non esiste una relazione padre-figlio, ma sono figli dello stesso tipo.

Delega ('delegation') Analogo a quello dei tipi laterali, questo meccanismo opera a livello delle istanze, ed è utilizzabile da linguaggi in cui il livello dei tipi non è considerato. In questo caso una variabile strutturata può essere creata facendole ereditare un certo numero di attributi (ed eventualmente il loro valore) da un'altra variabile.

Parametrizzazione È un meccanismo valido sia per gli aspetti di personalizzazione che per quelli di evoluzione. Essa permette di definire alcuni aspetti del modello del processo al momento della sua istanziazione, ma se usata opportunamente, anche durante la sua esecuzione.

I meccanismi sopraelencati si possono considerare espressione della genericità di cui si è parlato alcune pagine prima, che deve avere il linguaggio.

In genere è sufficiente che un linguaggio preveda la possibilità di utilizzare uno solo di questi meccanismi anche al fine di evitare una eccessiva complessità. Però può essere utile avere a disposizione due di questi, ad esempio uno più adatto alla personalizzazione e l'altro più adatto all'evoluzione.

Va tenuto presente che per la gestione dell'evoluzione di un prodotto, non solo è necessario l'uso di una base dati che supporti il versionamento, ma è bene che questa interagisca anche con un sistema di gestione delle configurazioni (*configuration management*). Considerando che, come già detto più volte, il modello del processo è del tutto analogo al prodotto software ed anch'esso evolve, questa integrazione può avere un doppio uso permettendo anche la gestione dei cambiamenti nel modello del processo.

2.5.2 Caratteristiche proprie di linguaggi per la modellizzazione dei processi

Fino ad ora si sono elencate caratteristiche presenti in linguaggi esistenti, anche se non tutte in uno stesso; se ne possono però individuare altre che sono prerogativa quasi unica dei linguaggi di modellizzazione dei processi.

Ambiguità Nella definizione di un modello di processo può essere utile avere la possibilità di dare *descrizioni informali* o addirittura *ambigue* oltre a quelle formali e precise. Infatti, come già detto in precedenza, la produzione di software è costituita da attività creative e quindi non completamente formalizzabili. Inoltre si deve essere in grado di descrivere, oltre alle più rigide e definite interazioni tra strumenti e macchine, anche quelle con le persone e tra le persone.

È evidente che un linguaggio che introduca ambiguità non si presta bene al raggiungimento di alcuni degli obiettivi della modellizzazione, quali ad esempio l'esecuzione o la verifica. Si tratta allora di raggiungere un valido

compromesso a seconda degli scopi che si intende effetivamente raggiungere con la modellizzazione.

Comprensibilità Il linguaggio per la modellizzazione dei processi deve essere *descrittivo* per aumentare la comprensibilità dei modelli in modo da recepirne i concetti fondamentali quasi a prima vista e con poche e semplici nozioni sul linguaggio di modellizzazione. Questo aumenta la capacità del modello di comunicare esperienze e metodologie tra gli addetti ai lavori e di estenderle anche a chi non lo sia.

Validazione e verifica È importante che il modello ottenuto possa essere facilmente sottoposto a *validazione e verifica*, così che si possa facilmente capire se esso vada presenti problemi o debba essere ottimizzato in alcuni suoi aspetti. Quindi il linguaggio utilizzato deve essere studiato in modo da facilitare queste operazioni.

Linguaggio modificabile durante l'esecuzione Come già ribadito più volte è necessario poter fare delle modifiche al modello del processo senza doverne interrompere e riprendere da principio l'attuazione. Un possibile approccio consta nell'esecuzione del modello in modo non convenzionale, cioè non basata sulla compilazione della descrizione nel linguaggio in questione. Ciò implica che questa venga interpretata o per lo meno compilata in moduli separati in modo da poter essere modificato senza doverne ricominciare l'esecuzione.

Ovviamente ci saranno restrizioni sui cambiamenti permessi in modo da mantenere la consistenza tra le varie entità coinvolte.

Meccanismi di suppporto per attività di gestione Le attività contemplate dal modello non devono essere solamente quelle tecniche, ma anche *attività di gestione* del processo quali:

- Gestione delle risorse con la possibilità di utilizzare strumenti che facilitino la scelta del miglior tipo di schedulazione tenendo presente la limitata disponibilità delle risorse.
- Pianificazione della suddivisione dei compiti tenendo in conto anche il carico di lavoro pendente sulle persone coinvolte. Anche per questo scopo è bene che l'ambiente metta a disposizione adeguati strumenti

in grado di mostrare le attività che devono essere assegnate per il normale svolgimento del processo o in seguito ad una deviazione dai piani precedentemente definiti.

- Controllo dell'avanzamento del processo nel suo complesso e delle singole attività. In questo caso possono essere utili strumenti che mantengono aggiornato il responsabile sull'avanzamento delle attività in corso, sulla conclusione di certe attività e sul fallimento di altre come anche sulla deviazione dai piani definiti in precedenza.

È quindi utile che il linguaggio utilizzato per la modellizzazione dei processi fornisca strumenti che facilitino la descrizione e l'esecuzione di queste attività.

3

Programmazione di processo

Uno degli obiettivi del lavoro descritto in questa trattazione, è la realizzazione di modelli di processo utilizzando un linguaggio di programmazione ad oggetti. Si tratta quindi di un caso particolare di programmazione di processo e per questo motivo in questo capitolo è trattato in modo particolarmente dettagliato questo approccio alla modellizzazione dei processi spiegandone i vantaggi. Sono descritte le peculiarità dei programmi di processo rispetto ai normali programmi e sono messe in risalto le caratteristiche che deve possedere un efficiente linguaggio per la programmazione di processo. Infine è fatta una panoramica dei principali ambienti PM basati su questo approccio.

3.1 Introduzione

La *programmazione dei processi* ('*process programming*') consiste nell'utilizzare normali linguaggi di programmazione per descrivere modelli di processi software sfruttando il fatto che i linguaggi per la definizione e manipolazione dei dati sono molto evoluti e largamente diffusi.

Si è già più volte evidenziata l'analogia esistente tra processi software e prodotti software (programmi) per cui può essere un passo piuttosto immediato quello di pensare di descrivere i processi esattamente come si descrivono i prodotti software.

Inoltre sia i processi software che i loro prodotti sono una descrizione di azioni da intraprendere (da parte di uomini e macchine nel primo caso o da parte di sole macchine nel secondo caso). In entrambe i casi queste azioni sono volte ad un ben preciso fine.

Anche il modello di processo può essere visto come eseguibile e, anche se tale esecuzione ('enactment') richiede l'interazione di macchine e uomini, è pur sempre realizzata mediante istanziazione e collegamento ('binding').

Infine i linguaggi di programmazione, se utilizzati per descrivere processi software, permettono di modellizzare processi di qualsiasi tipo e a differenti livelli di astrazione.

Si possono notare differenze nel fatto che gli ingressi e le uscite dei normali programmi sono dati di piccole dimensioni mentre quelli dei processi software sono più grandi.

3.2 Vantaggi della programmazione di processo

Rispetto ai metodi normalmente utilizzati in altri campi della produzione per la desccrizione dei processi (manuali di procedura o diagrammi di Pert), un approccio di questo genere è senza dubbio più *completo e rigoroso*.

Il vantaggio che si può trarre dall'uso di un normale linguaggio di programmazione sta soprattutto nel fatto che esso sarà già *noto* agli sviluppatori del software e quindi i modelli saranno più facilmente comprensibili. Questo facilita uno dei compiti della modellizzazione dei processi che deve essere la diffusione e *comunicazione* di una certa quantità di conoscenze.

Come il linguaggio rappresenta un veicolo di diffusione di informazioni tra gli uomini, così esso può essere mezzo di comunicazione tra macchine e persone permettendo di *attuare* i processi con l'aiuto del calcolatore.

Poiché il linguaggio ha una ben precisa sintassi e semantica c'è controllo automatico della *correttezza e coerenza* del moello del processo.

Inoltre il linguaggio permette di vedere il problema a *diversi livelli di astrazione* usando chiamate a procedura che al livello superiore liberano dai dettagli implementativi dei livelli inferiori del modello.

Perché il processo software sia eseguibile, è necessario che tutte le operazioni siano specificate fino al livello più basso in modo che la macchina le possa eseguire o fino ad un livello atto ad essere eseguite da strumenti automatici o da persone.

Il modello del processo, o meglio in questo caso il *programma del processo* ('*process program*'), può essere considerato l'indicazione di come gli

strumenti, integrati con le risorse umane, devono attuare i processi. Le entità tipiche della modellizzazione dei processi sono associate ad entità tipiche della programmazione. Gli oggetti software sono pensati come variabili e quindi istanze di tipi. Gli strumenti sono operatori su questi oggetti che li trasformano; analogamente fanno gli uomini cui sono affidati certi ruoli con compiti ben definiti.

Non è detto che il linguaggio utilizzato debba essere procedurale, ma può essere basato su regole ('rule-based') oppure ad oggetti; l'unico aspetto che distingue questo approccio da quelli descritti nel capitolo successivo sta nel fatto che i formalismi utilizzati in questo caso sono normali linguaggi di programmazione esistenti sul mercato.

È importante vedere il modello del processo come un programma software perché si può così meglio esplicitare l'affermazione fatta più volte secondo cui "i modelli di processi software sono anch'essi software" [Ost87]. Inoltre si possono applicare al 'process program' tutti i metodi e le procedure normalmente applicati ai programmi tradizionali.

3.3 Peculiarità dei programmi di processo

La principale differenza tra la programmazione dei processi ('process programming') e la normale programmazione sta nel differente dominio in cui sono utilizzate. Innanzitutto gli oggetti trattati dalla prima sono di dimensioni maggiori rispetto a quelli normalmente trattati dalla seconda, e soprattutto sono definiti in modo meno preciso perché più difficilmente comprensibili.

Molto importante è il fatto che i prodotti dell'esecuzione dei programmi di processo ('process programs') non sono semplici dati passivi, ma altre definizioni di processi. Cioè il prodotto software ottenuto è un processo che istanziato ed eseguito consente all'utente finale di raggiungere gli scopi richiesti mediante le specifiche date per il progetto [Ost87].

Il prodotto software comprende, oltre ai moduli di codice che rappresentano la descrizione eseguibile del programma, anche i sorgenti, la documentazione, i vari documenti di specifica, analisi e progetto, i test, ecc. Analoga è la descrizione del processo software secondo il paradigma del 'process programming'. Il programma di processo, creato dall'ingegnere del software, interagisce con il professionista del software, mentre il programma finale interagisce con l'utente.

Bisogna considerare l'ipotesi di definire anche processi per la produzione

ed evoluzione di modelli di processi (*meta-processi*). Ciò può far pensare di giungere ad avere una gerarchia senza fine di processi per trattare processi che si complica sempre più. Questo aumento di complessità è però evitato grazie al fatto che i processi hanno stessa struttura del loro prodotto e quindi possono essere trattati con le stesse tecniche e metodologie.

3.4 Caratteristiche di un linguaggio per programmazione di processo

Per la modellizzazione dei processi devono essere disponibili costrutti linguistici potenti e soprattutto, poiché servono strutture dati molto complesse, sono indispensabili efficienti meccanismi di definizioni di tipi e di aggregazione di dati. Non meno importanti sono strutture flessibili e potenti per il controllo del flusso.

Come detto nel capitolo precedente, un linguaggio di modellizzazione dei processi, anche se non è un normale linguaggio di programmazione, deve supportare la concorrenza dal momento che i processi software sono intrinsecamente concorrenti perché si hanno vari strumenti automatici e persone che agiscono contemporaneamente per portare avanti lo sviluppo del prodotto. Inoltre si ha interazione tra queste entità per cui sono necessari potenti e sicuri meccanismi per lo scambio di messaggi.

È bene disporre di un ambiente per la scrittura e l'esecuzione dei programmi di processo che sia completo degli strumenti necessari. Questi si possono vedere, dal punto di vista della programmazione dei processi, come operatori che agiscono sugli oggetti dei processi software, cioè sulle istanze dei tipi dichiarati all'interno del 'process program' mediante il linguaggio scelto.

Come i prodotti software e gli elementi che permettono di definire i processi per generarli, si possono vedere come oggetti in questo ambiente, così anche i processi software possono essere visti come istanze di opportuni metatipi creando una gerarchia di tipi corrispondente alla gerarchia di processi di cui si è parlato alla fine della sezione precedente.

3.5 Ambienti PM esistenti

Come varrà più volte sottolineato in seguito non esistono sistemi che facciano uso di un singolo paradigma di modellizzazione dei processi. Analogamente i prodotti di seguito presentati non usano un vero e proprio linguaggio normalmente utilizzato per programmare, ma una sua estensione.

APPL/A [Jr.90] è il prototipo di linguaggio per la programmazione dei processi utilizzato nel progetto Arcadia. È un'estensione di Ada che permette la definizione di relazioni tra gli oggetti che possono essere derivative o attive, cioè attivate da un'operazione. Inoltre le attivazioni propagano i cambiamenti da una relazione all'altra e si possono definire predicati sulle relazioni.

Essendo basato su Ada, ne eredita le caratteristiche principali come il sistema dei tipi, lo stile di definizione dei moduli e i metodi di comunicazione tra le varie attività che seguono il meccanismo del 'rendezvous'.

Le descrizioni dei processi sono date secondo un approccio procedurale, ma si possono usare anche metodologie basate su regole ad esempio per specificare condizioni di consistenza.

Il supporto ai cambiamenti del processo, e quindi alla loro evoluzione e configurazione, è debole.

OPM **Object Process Modeling Environment** [YE89] fornisce un supporto per progettare ed eseguire modelli di processi software. I modelli di processo sono scritti in **Galois** [Sug90] che è un linguaggio ad oggetti per la programmazione dei processi e sono eseguiti dal sistema di supporto all'esecuzione di OPM [YE90].

Galois è un'estensione del C++, ma da questo differisce per quattro principali innovazioni.

1. Utilizza le *metaclassi*, cioè prevede una rappresentazione esplicita e direttamente modificabile delle classi. Questo è utilizzato perchè i modelli di processo sono definiti come classi e i processi sono ottenuti istanziando oggetti di queste classi.

2. Supporta la *derivazione* ('*delegation*' [Ste87]), cioè gli oggetti, pur appartenendo ad una certa classe, possono anche non essere creati dalla loro classe, ma da un altro oggetto.

3. Considera le operazioni eseguibili sugli oggetti (cioè i loro metodi) come *oggetti tipati* e la classe di cui sono istanze determina il tipo di schedulazione cui sono soggette. Infatti ogni oggetto ha un solo flusso di esecuzione e reagisce alle chiamate dei suoi metodi eseguendo la prima ed accodando le successive (comportamento tipico dei *monitor*). La schedulazione delle chiamate accodate è fatta utilizzando un algoritmo differente a seconda della classe dell'operazione invocata.

4. Aggiunge caratteristiche tipiche dell'*approccio a regole* permettendo di associare ai metodi precondizioni e postcondizioni che ne detrminano l'eseguibilità.

IPSE 2.5 [War89] ha un liguaggio di programmazione dei processi concorrente ed imperativo.

4

Altri approcci ed ambienti esistenti

Per la modellizzazione dei processi sono stati teorizzati molti approcci differenti, ma nessuno è considerato essere sufficiente. Inoltre in molti prodotti attualmente esistenti vengono utilizzati congiuntamente più metodi per cui è anche difficile fare una classificazione di questi ambienti. In questo capitolo sono esposti i principali approcci alla modellizzazione dei processi ed i rispettivi ambienti PM classificandoli in base all'approccio principalmente e più diffusamente utilizzato; alcuni prodotti sono stati classificati come ibridi.

La maggior parte degli ambienti elencati di seguito non sono effettivamente utilizzati per la produzione, ma sono prototipi realizzati a livello accademico ed in campo di ricerca.

4.1 Tecniche di intelligenza artificiale e basate su regole

I metodi basati su regole ('rule-based systems') modellizzano i processi mediante insiemi di regole che agiscono su una base di conoscenza ('knowledge base'). La stessa tecnica è utilizzata anche per eseguire il processo oltre che per descriverlo.

Le regole sono costituite da una precondizione, un corpo ed una postcondizione. Quando la precondizione è verificata viene eseguito il corpo, che è in genere un'attività o una parte di questa, dopo di che almeno una delle postcondizioni risulta vera. L'esecuzione modifica la base di conoscenza per

cui può far avverare precondizioni di altre regole e quindi attivarle.

Le diverse realizzazioni di sistemi basati su regole hanno differenti modi di strutturare la base di conoscenza e di eseguire le regole, in particolar modo quando più di una sia contemporaneamente attiva, fornendo o meno meccanismi di tracciamento all'indietro ('backtracking'). Questi vengono utilizzati quando una certa configurazione della base di conoscenza attivi diverse regole nello stesso istante; dopo averne eseguita una, probabilmente le precondizioni delle altre non sarebbero più verificate, però si eseguono ugualmente tutte ad una ad una.

In alcuni casi il processo è visto come un insieme di compiti tra loro collegati ed espletati da una serie di agenti intelligenti e piuttosto indipendenti (debolmente accoppiati) che possono modificare la base di conoscenza agendo in modo diverso a seconda di cosa vi è contenuto. Tutte le loro azioni ed interazioni sono uno sforzo intelligente per produrre nel modo migliore e con l'uso di risore più conveniente.

A seconda dell'implementazione considerata vengono più o meno pesantemente utilizzate tecniche proprie dell'intelligenza artificiale per gestire l'attuazione delle varie regole e le loro interazioni.

Questo paradigma presenta alcuni problemi derivanti in primo luogo dalla scarsa attitudine dei classici linguaggi basati su regole nell'esprimere transazioni a lungo termine ('long transactions'). Inoltre le regole e le tecniche dell'intelligenza artificiale da sole non sono in grado di controllare transazioni cooperative sulla base dati o la gestione delle configurazioni e delle versioni. L'evoluzione stessa può essere gestita mediante ripianificazione e riesecuzione [LC93].

Problemi possono nascere dal fatto che spesso non è noto a priori l'ordine in cui le regole possono essere eseguite se non utilizzando precondizioni molto elaborate. Se da un lato questo fatto permette di modellizzare bene il non determinismo, d'altro canto in alcune situazioni può essere dannoso non sapere a priori cosa accade quando una certa condizione può attivare diverse regole.

Di contro c'è il vantaggio di poter facilmente modificare il modello del processo in modo dinamico semplicemente cambiando, aggiungendo o eliminando regole, oppure il contenuto della base di conoscenza.

4.1.1 ALF

Utilizza un approccio fondato su una base di conoscenza ed affronta i problemi derivanti dalla personalizzazione. Il modello del processo è descritto da una gerarchia di MASP [B+89] ('Model for Assisted Software Process'): infatti ogni MASP può essere espresso in dettaglio da un sottoinsieme di MASP così che è possibile fornire una descrizione a diversi livelli di astrazione.

Ogni MASP è costituito da sei elementi.

- Il *modello degli oggetti* ('*object model*') che fornisce un modello di dati concettuale basato su un approccio entità-relazioni-attributi.
- Un insieme di *espressioni* in un linguaggio di tipo a regole del primo ordine.
- *Tipi di operatori* che descrivono la semantica delle attività del processo software mediante precondizioni e postcondizioni.
- *Regole* che definiscono le possibili reazioni automatiche che si possono avere in seguito al verificarsi di certe condizioni durante l'attuazione del processo.
- *Caratteristiche* che servono a specificare delle restrizioni sugli stati in cui può venirsi a trovare il processo. Se una non è rispettata, viene scatenata un'eccezione.
- *Ordini*, cioè una serie di 'path expressions' che definiscono l'ordine tra le varie azioni specificando quali possono essere eseguite in parallelo, quali sequenzialmente e quali alternativamente.

Uno degli obiettivi principali di ALF è di fornire assistenza durante lo sviluppo del software. L'utente è guidato dicendogli ciò che deve fare, l'ordine con cui farlo e come vanno portate a compimento certe azioni. Inoltre l'utente riceve spiegazioni quando il sistema prende un'iniziativa o quando una certa operazione iniziata dall'utente viene rifiutata.

Il MASP è un modello di processo generico che può essere personalizzato in un modello specifico del progetto detto IMASP. Ciò porta alla istanziazione dei sei componenti del MASP precedentemente elencati con un meccanismo basato sulla sostituzione di opportuni valori ai loro parametri formali. Quindi

il problema della personalizzazione in ALF è risolto mediante l'utilizzo della parametrizzazione .

Per attuare il processo è infine necessario istanziare l'IMASP ottenendo un ASP che è il vero e proprio processo per la produzione di un ben specifico pacchetto software.

4.1.2 MARVEL

Nell'ambiente MARVEL [BK92] i modelli dei processi sono espressi sottoforma di regole che sono prese da tre insiemi.

- Insieme delle regole del progetto ('project rule set') i cui elementi servono per descrivere problemi tipici del processo. Queste comprendono due sottoinsiemi di regole:
  - regole di inferenza ('inference rules') che servono a definire relazioni tra gli attributi degli oggetti;
  - regole di attivazione ('activation rules') che provvedono a far iniziare le attività di sviluppo necessarie per far avanzare il processo. Queste regole in genere sono costituite dall'invocazione di uno strumento automatico ed hanno un certo numero di effetti mutuamente esclusivi a seconda di come viene portata a termine l'attività controllata.
- Insieme dei tipi del progetto ('project type set') i cui elementi sono utilizzati per specificare, con una metodologia ad oggetti i dati trattati.
- Insieme degli strumenti del progetto ('project tool set') i cui elementi permettono di rappresentare le interacce con gli strumenti automatici.

Le regole possono essere eseguite con una modalità '*forward chaining*' o '*backward chaining*'. La prima consta nell'aspettare che tutte le condizioni di una regola siano verificate per eseguire tale regola. La seconda invece consta nell'agire, appunto all'indietro, cercando di attivare forzatamente (e quindi utilizzando in modo ricorsivo questo metodo) altre regole che potrebbero rendere vere le condizioni necessarie ad innnescare la regola voluta. Se non si riesce in alcun modo a far verificare le precondizioni si notifica di ciò chi ha chiesto il 'backward chaining', diversamente si esegue la regola che magari rende vere le condizioni che attivano altre regole con un meccanismo 'forward chaining'.

4.1.3 MERLIN

MERLIN [EJP+91] è un ambiente per la modellizzazione dei processi che si appoggia ad un linguaggio in cui si utilizzano tecniche basate su regole per costruire e operare su una base di conoscenza. Il modello del processo è descritto con un insieme di regole che sono molto simili a quelle utilizzate nel linguaggio Prolog.

Quando un utente comincia a lavorare sono recuperate da GRAS, che è una base dati non convenzionale in grado di gestire grafi, alcune regole che descrivono le azioni che tale utente deve fare a seconda di quello che è il suo ruolo.

Le regole possono essere eseguite secondo modalità 'forward chaining' o 'backward chaining'. Quest'ultimo meccanismo è usato per selezionare i ruoli e le attività che in un certo momento sono disponibili per un certo utente, oppure per raccogliere informazioni sullo stato del processo. Invece le regole 'forward chaining' sono prevelentemente utilizzate quando occorre che il sistema fornisca una guida esplicita all'utente. Cioè, in base all'obiettivo da raggiungere, vengono fornite le attività necessarie.

Infatti ogni utente ha più ruoli ed ogni ruolo ha un certo numero di attività che possono essere abilitate o meno. Gli utenti sono guidati nella scelta degli uni e delle altre; quando una persona si collega al sistema gli viene mostrato un certo numero di ruoli al momento attivi, tra cui egli può scegliere. Una volta scelto il ruolo gli vengono proposte le attività che in quel momento sono disponibili per esso. Infine l'utente viene guidato nello svolgimento dell'attivià che ha scelto.

Tutte le informazioni utili sono memorizzate nella base di conoscenza e lo stato del progetto è aggiornato in tempo reale in seguito alle attività svolte dall'utente cui corrispondono manipolazioni alla base di conoscenza. Così, poiché i fatti e le regole possono essere dinamicamente inseriti e cancellati dalla 'knowledge base', il modello del processo è molto flessibile ed è possibile apportare ad esso cambiamenti in modo dinamico.

4.1.4 Oikos

Oikos [ACM90] è un ambiente per la modellizzazione dei processi software basato sulla programmazione logica e sul paradigma della lavagna ('blackboard paradigm').

Uno dei concetti basilari di Oikos è quello di *entità atomica*: un sistema

reattivo modulare utilizzato per modellizzare le risorse tipiche di un processo per lo sviluppo del software. Ci sono poi le *entità composite* che servono per modellizzare le attività necessarie per portare avanti i processi software. Le varie entià possono essere decomposte in modo gerarchico in sottoentità fornendo desccrizioni a diversi livelli di astrazione [ABGM92].

Altro concetto importante è quello degli *agenti* che sono associati alle varie 'blackboards' e si comportano come sistemi reattivi eseguendo certe azioni in base ai fatti contenuti nella lavagna cui sono associati. Il loro comportamento è guidato da una *teoria* ('*theory*'), cioè un insieme di modelli di reazione ('reaction patterns') e di regole Prolog. Ciascun modello di reazione è costituito da un insieme di fatti di attivazione e da un'azione in risposta. Quando nella 'blackboard' associata sono presenti i fatti si attivazione, l'agente esegue le operazioni predefinite e inserisce nella sua lavagna un insieme di fatti detto 'success set'. Nel caso la sua azione non termini correttamente, esso inserisce nella propria lavagna altri fatti raggruppati in un 'failure set'.

I vari agenti vengono eseguiti in modo concorrente leggendo contemporaneamente le informazioni nelle loro lavagne e facendo il loro compito non appena sono presenti i fatti di attivazione. Quando più 'reaction patterns' divengono attivi, quello da eseguire è scelto in modo non deterministico e senza alcun meccanismo per tornare a considerare anche gli altri ('backtracking'). I vari agenti possono comunicare tra loro scrivendo uno nella lavagna associata all'altro.

Importante e singolare meccanismo presente in Oikos è quello dell'*angelo*. Una o più di queste entità possono essere associate ad ogni 'blackboard' ed hanno la caratteristica di inserire fatti in essa esattamente quando questi servono. Cioè gli angeli non sono attivati da ciò che è contenuto in un certo istante nella lavagna, ma fanno sì che il contenuto sia istante per istante quello necessario affiche tutto funzioni in modo corretto ed efficiente. Nonostante un tale concetto possa sembrare strano, questi particolari agenti divengono molto utili nella modellizzazione di certi aspetti del dominio in esame.

Tutti i documenti coinvolti nel processo di produzione del software sono contenuti in una base dati che l'ambiente Oikos fornisce come sevizio. Il sistema di gestione della base dati offre alcuni schemi predefiniti contenenti la struttura di documenti correlati ad Oikos e permette la creazione di nuovi schemi. Oltre ai documenti è possibile memorizzare la struttura del processo stesso.

La base dati fornita è di tipo logico-deduttiva, cioè in grado di elaborare

interrograzioni deduttive, nel senso che permette di ricavare nuovi fatti da quelli presenti e non solo di mostrare questi ultimi.

4.2 Basi dati attive

Come nella quasi totalità dei sistemi, gli ambienti che utilizzano questo paradigma si basano su un sistema di gestione di una base dati che in questo caso non ha però solamente una funzione di substrato, ma un ruolo di maggior rilievo.

Questo sistema di gestione comprende regole di tipo evento-condizione-azione: quando si verifica l'*evento* specificato, che è generalmente una operazione sulla base dati, viene valutata la *condizione* associata e, se questa è trovata vera, viene eseguita l'*azione* corrispondente. Questa a sua volta può innescare altre regole o perchè modifica delle condizioni o perchè attiva degli eventi.

Si tratta comunque di sistemi di modellizzazione dei processi a basso livello che supportano solamente transazioni brevi e attività di basso livello.

Il problema delle regole ECA, come è stato detto per i sistemi basati su regole, è che non sono in grado di esprimere facilmente operazioni lunghe, ritardate o privilegiate. Inoltre sono carenti nel descrivere le attività di alto livello eseguite da persone, la gestione degli errori e la pianificazione generale, nonchè nel definire l'ordine in cui le azioni vanno eseguite.

La personalizzazione del modello del processo dipende dalla struttura delle regole e il cambiamento delle regole, per realizzare l'evoluzione del modello del processo, può essere pesante da gestire. Inoltre a seconda della base dati sottostante si avrà una gestione delle versioni e delle configurazioni più o meno efficiente.

4.2.1 Adele

Adele [BEM91] è un ambiente costruito su una base dati versionata fondata su un modello entità-relazione che è stato esteso secondo l'orientamento ad oggetti e che gestisce transazioni di lunga durata e l'uso di comandi ed azioni definiti dall'utente. Le azioni che hanno lunga durata non sono fatte direttamente sulla base dati, ma su un suo sottoinsieme detto contesto di lavoro ('workcontext') che è associato ad un certo utente e che comprende un insieme di strumenti, documenti e attività da svolgere.

L'innescamento degli eventi tipici di meccanismi ECA è determinato da operazioni sulla base dati ed è utilizzato per consentire alcune azioni specifiche del processo software come il controllo dei vincoli di integrità e la propagazione delle modifiche. Ad esempio se viene modificata l'interfaccia di un modulo, questo meccanismo permette di far automaticamente scattare una serie di azioni che devono seguire una tale modifica (come la ricompilazione di certi moduli) per mantenere la coerenza del prodotto.

Lo scopo di Adele non è tanto di permettere una completa formalizzazione dei modelli di processo, ma di aiutare il controllo dei processi software.

4.2.2 MVP-L ('Multiview Process Modeling')

MVP-L [Rom91] è un linguaggio testuale che vuole fornire un mezzo per costruire, analizzare, eseguire e migliorare modelli descrittivi di processi software di grandi dimensioni e reali. Particolare attenzione è posta nel miglioramento dei processi fornendo misure rilevate durante l'attuazione. Importante è considerato anche il riutilizzo facilitato mediante una particolarmente curata divisione in moduli ('packaging').

I modelli di processo generici sono descritti facendo uso della tipizzazione ('typing'). I processi specifici si ottengono poi istanziando questi tipi.

Inoltre sono fornite definizioni di processi elementari e modelli di prodotti e di risorse che possono essere utilizzati come mattoni elementari nella realizzazione di modelli di processi.

4.3 Automi e reti

Questo paradigma nasce dalla considerazione che un processo software è simile ad un sistema in tempo reale ('real-time system') per cui si utilizzano tecniche simili a quelle utilizzate per creare e far funzionare sistemi in tempo reale come ad esempio reti di Petri potenziate.

Queste permettono di modellizzare molto bene il parallelismo ed il non determinismo che, come affermato in precedenza, sono aspetti molto importanti nella modellizzazione dei processi. Altro vantaggio non trascurabile, anche se meno determinante, sta nel fatto che le reti forniscono un approccio formale e visuale che può spesso essere molto utile e chiarificante.

Caratteristica saliente di questa metodologia è che non si definisce un modello di processo generico, ma direttamente il modello del processo specifico

del progetto che è costituito da una particolare rete o grafo. L'attuazione del modello del processo consta nell'innescamento delle transizioni e loro esecuzione.

Nel modello derivante dall'uso di reti di Petri modificate, le attività non sono organizzate gerarchicamente (suddivise in sottoattività), ma solamente si dà l'ordine in cui queste devono essere portate a termine. Inoltre tale modello è impostato come un insieme di entità che rispondono a degli stimoli ('reactor-oriented') provenienti da un misto di persone e strumenti automatici, eseguendo ognuna una attività in modo concorrente con le altre.

Questo approccio presenta dei problemi derivanti dalla sua rigidità, sia perchè non sono diffusi meccanismi di collegamento a posteriori ('late binding'), sia perchè la personalizzazione ed evoluzione del processo sono difficilmente realizzabili. Inoltre i sistemi basati esclusivamente sulle reti o sugli automi non forniscono un efficiente supporto per la gestione delle configurazioni e delle versioni.

4.3.1 DesignNet

DesignNet [LH89] fornisce un ambiente per una efficiente gestione dei progetti per la produzione del software permettendo di definirne misure della qualità. Il risultato di queste misure può a sua volta essere utilizzato per migliorare tale gestione.

I processi sono descritti mediante reti di Petri arricchite di grafi AND/OR che permettono così di rappresentare la struttura gerarchica dell'insieme delle attività. Le reti costituiscono il modello a livello dei tipi, mentre l'istanza del modello del processo è rappresentata da reti con i vari gettoni ('tokens'), che regolano il flusso delle operazioni, posizionati ('marked nets').

L'ambiente comprende anche una base dati di supporto che si appoggia a VBase fornendo una rappresentazione ad oggetti di tutte le entità presenti in DesignNet e delle loro relazioni. Una caratteristica di rilievo è che i vari documenti sono associati ad una informazione di tempo ('time stamp') e si afferma che è fornita una valida gestione delle configurazioni e delle versioni.

4.3.2 Entity Process Model (EPM)

EPM realizza modelli di processo in cui, invece che considerare le attività che devono essere eseguite per portare a compimento la produzione, focalizza

l'attenzione sulle *entità* che intervengono nel processo software e su cui le suddette attività operano. Queste entità sono tali solo se hanno un ben determinato periodo di vita nell'ambito del processo e tale periodo deve avere la durata di tutto il tempo di vita utile del sistema software [HK89]. Tipiche entità sono i documenti delle specifiche, i progetti, il prodotto finito, la sua documentazione.

Sulle entità agiscono le attività facendole passare attraverso una serie di *stati*. La distinzione tra entità ed attivtà è volta a differenziare i prodotti del processo software dai meccanismi utilizzati per controllarlo e portarlo a termine.

Si considera che le entità permangano in un certo stato per un tempo non nullo. In tale stato le entità possono essere create o modificate (*stato attivo*) o rimanere nelle stesse condizioni (*stato passivo*). Ogni stato può essere ulteriormente suddiviso in un insieme di sottostati eventualmente eseguibili in parallelo o essere atomico. Si può quindi realizzare una visione gerarchica degli stati con diversi livelli di astrazione. Le transizioni sono supposte di durata trascurabile e sono scatenate da una serie di condizioni che si avverano.

Il modello del processo è allora rappresentato come un automa a stati finiti e per questo vengono utilizzate come supporto alla modellizzazione gli strumenti messi a disposizione da STATEMATE [H$^+$88]. Da questo fatto deriva però un problema perchè STATEMATE non possiede il concetto di entità: queste vengono quindi rappresentate come componenti ortogonali (eseguibili in parallelo) di uno stato al più alto livello di astrazione. Ognuno di questi componenti sarà poi diviso in una serie di sottostati che descrivono l'evoluzione dell'entità stessa.

Gli strumenti di STATEMATE sono utilizzati anche per pianificare l'utilizzo delle risorse e l'analisi dei processi modellizzati. Infatti in primo luogo si può tracciare un grafico dell'andamento temporale del processo in cui per ogni istante si indica in quali stati si trovano le varie entità e quali risorse utilizzano. Questo grafico è ottenuto realizzando le transizioni da uno stato all'altro appena possibili senza tenere conto della disponibilità delle risorse che uno stato può richiedere. Per questo motivo il modello che genera tale diagramma è detto modello del processo non vincolato ('*Unconstrained Process Model*').

In seguito, basandosi su tale grafico per individuare i periodi in cui si

utilizzano più risorse di quelle disponibili, si possono proporre differenti evoluzioni attraverso gli stati facendo in modo di non fare le transizioni che portano in stati che richiedono troppe risorse. Questo dà origine ad una serie di diagrammi alternativi che nascono da possibili scelte di transizioni che richiedono però un diverso numero o tipo di risorse. Queste scelte corrispondono ad un modello di processo così detto vincolato ('*Contrained Process Model*') e danno origine a processi di durata differente. Allora analizzando questi diagrammi è possibile stabilire quale sia la migliore alternativa in base a ben determinate politiche e preferenze.

EPM non prevede l'appoggio di un sistema di gestione di una base dati.

4.3.3 Reti FUNSOFT

Si tratta di una modellizzazione basata sull'uso di reti di Petri di alto livello costruite su reti di tipo Predicato/Transizione. Si definisce infatti una estensione di queste mediante l'aggiunta di caratteristiche considerate utili nella descrizione dei processi. Tra queste ad esempio la possibilità di associare delle politiche alle piazze e di supportare differenti comportamenti di attivazione a seconda del numero di 'token' prodotti o consumati [ABGM92].

Utilizzando un formalismo basato sulle reti di Petri, le reti FUNSOFT sono in grado di rappresentare in modo semplice e naturale il non determinismo ed il parallelismo. Gli oggetti che intervengono nel modello del processo sono rappresentati con un formalismo ad oggetti in cui l'insieme di oggetti utilizzabili può essere esteso.

A differenza di EPM, le reti FUNSOFT non considerano che le transizioni avvengano in tempo nullo, ma è possibile associare loro un valore per modellizzare la durata dell'attività rappresentata da quella transizione. Questo fatto può così essere utilizzato in fase di simulazione per dare un andamento più simile alla realtà del comportamento del processo descritto dalla rete.

Le reti FUNSOFT sono particolarmente adatte alla rappresentazione di attività complesse. Particolare attenzione è stata posta all'aspetto della simulazione e della valutazione della validità del processo. Esse prevedono la modifica dinamica del processo, però l'attuazione del modello del processo non è supportata completamente.

MELMAC/MSP [DG90] è un prodotto che fornisce due livelli di descrizione per i processi software. Ad alto livello esso fornisce viste dei tipi di oggetto, le attività, i processi e la gestione del progetto. A basso livello le

entità sono medellizzate mediante reti FUNSOFT.

4.3.4 SPADE

SPADE [BFG91] è un ambiente per la modellizzazione dei processi software che possiede un linguaggio per la descrizione dei processi chiamato SLANG che è basato su reti di Petri ad alto livello dette reti ER.

Si ha un concetto distribuito di stato che è fornito dal piazzamento dei 'token' nella rete. La topologia della rete stabilisce l'ordine in cui devono essere eseguite le varie attività e quelle che possono essere portate avanti in parallelo. Inolre permette di risolvere eventuali situazioni di conflitto.

Le piazze modellizzano dati, strumenti o risorse e la presenza del 'token' sulla piazza indica la disponibilità dell'entità rappresentata. Invece le transizioni rappresentano eventi che possono accadere quando siano vere le condizioni descritte dalle piazze di ingresso (cioè quando ognuna ha il suo 'token'). Le azioni da intraprendere quando si verifica un evento sono descritte con un linguaggio di tipo logico.

Si possono rappresentare anche informazioni di tipo temporale come ad esempio il tempo entro cui un evento deve accadere, associando ai 'token' un'indicazione temporale ('time-stamp') e alle azioni un avanzamento del riferimento temporale.

SLANG è un linguaggio altamente espressivo perchè permette di rappresentare in modo omogeneo differenti aspetti dei modelli di processo come la gestione delle risorse umane, le relazioni di precedenza tra gli eventi ed informazioni di tipo temporale.

4.4 Contratto dinamico e ISTAR

ISTAR [Dow87] è un ambiente di supporto ai progetti basato su un approccio contrattuale ed è l'unico noto che lo utilizza. Questo metodo vede ogni attività all'interno del modello del processo come un contratto tra uno stipulatore ed un cliente. Il contratto viene visto come l'insieme dei documenti di ingresso e di uscita, i prodotti da consegnare, i test di accettazione e le politiche di lavoro.

Il modello di processo globale è descritto come una gerarchia di contratti che vengono creati dinamicamente dagli sviluppatori del progetto che, a seconda dei momenti, si comportano come stipulatori o come clienti.

Come accade in altri campi dell'ingegneria, nell'approccio di ISTAR il processo di produzione è guidato dalla struttura del prodotto. Però con questa metodologia non c'è una modellizzazione formale del processo software e il tutto è ridotto alla gestione del protocollo tra stipulatore e cliente.

4.5 Grammatiche ad attributi e HFSP

HFSP ('Hierarchical and Functional Software Process description and enaction') [Kat89] utilizza, nel modellizzare il processo, un approccio funzionale basato sulle grammatiche ad attributi. Il concetto di base è che le attività sono funzioni che, presi alcuni oggetti in ingresso, ne producono altri in uscita. Le attività sono gerarchicamente decomposte usando regole di grammatica.

Il nome grammatiche ad attributi sta ad indicare che alle regole di grammatica sono associati degli attributi che rappresentano gli ingressi e le uscite. Questi collegano dunque le attività tra loro perchè le une utilizzano come ingresso le uscite delle altre o viceversa.

Inoltre, come ogni altro linguaggio funzionale, HFSP è *riflessivo*, cioè tratta lo stato dell'esecuzione come un qualunque tipo di dato e lo può quindi manipolare. Ciò implica che il processo può essere modificato da se stesso durante la propria esecuzione.

HFSP è intrinsecamente concorrente perchè , a meno che non ci siano dipendenze tra gli attributi, essi possono essere valutati in parallelo. Si può anche esprimere il non determinismo fornendo due regole per descrivere la stessa attività.

Il linguaggio può anche interagire con una base dati da cui vengono prelevati gli oggetti in ingresso alle attività ed in cui vengono riposti gli oggetti in uscita. Non è ben chiaro come possano essere modellizzate e gestite le interazioni delle persone tra loro e con gli strumenti.

4.6 Approccio ibrido

Come detto in precedenza una singola metodologia non è sufficiente per la modellizzazione dei processi. Tutti i sistemi presentati precedentemente non seguono un solo paradigma, ma si possono ugualmente classificare in base a quello maggiormente sfruttato. Altri non sono chiaramente classificabili e per questo li trattiamo come caso a parte.

Un chiaro esempio è **SPECIMEN** [Sch90] che unisce le reti FUNSOFT al linguaggio di modellizzazione dei processi basato su regole MERLIN per costruire un proprio linguaggio di modellizzazione dei processi.

Epos

Epos [CWL+89] si basa su un modello oggetti-relazioni in cui gli oggetti hanno stato e comportamento (cioè forniscono dei servizi) e comunicano mediante passaggio di messaggi. Le classi sono esplicitamente rappresentate, cioè il sistema è *riflessivo* perchè in grado di manipolare le definizioni stesse delle classi che utilizza. Inoltre le varie classi sono organizzate gerarchicamente e tra esse vi è ereditarietà.

Le descrizioni dei processi e dei dati sono memorizzate in una base dati organizzata secondo un modello entità-relazione detta EPOSDB. Per questo motivo nella base dati vengono memorizzati solo gli attributi degli oggetti e le relazioni tra di essi; i metodi sono scritti nel linguaggio SPELL e memorizzati a parte. Anche l'esecuzione dei metodi avviene all'interno di SPELL.

Nella base dati, oltre alle classi e gli oggetti che compongono il modello del processo, sono contenuti anche i prodotti. Un processo è modellizzato come un insieme di attività che possono essere create dinamicamente.

EPOSDB supporta transazioni lunghe, cooperative ed annidate. Inoltre si ha in Epos un'organizzazione gerarchica dei progetti in cui i progetti figlio ereditano il sottoinsieme della base dati del padre che possono personalizzare. Ogni progetto è modellizzato in Epos associandogli gli strumenti, i ruoli ed il modello di processo generico che esso utilizza.

Nell'ambiente è presente un PM Manager che permette di gestire la modellizzazione dei processi. Questo consente di sviluppare due fasi del meta-processo di modellizzazione: la fase di progettazione del modello del processo che consta nel definire le classi, gli oggetti e le relazioni che costituiscono il modello, e la fase di personalizzazione che consta nel raffinare le entità precedentemente definite. Proprio perchè si tratta in entrambe i casi di manipolazioni di classi, per entrambe le attività si fa uso dello stesso strumento.

Per l'istanziazione del processo si utilizza un altro strumento che è il Planner che basa il suo funzionamento su tecniche di intelligenza artificiale. L'istanza generata è poi eseguita dall'Execution Manager che schedula i processi, ne valuta le precondizioni e stabilisce l'ordine di esecuzione.

Tutte le varie fasi che fanno parte del meta-processo utilizzato in Epos

per la modellizzazione dei processi, possono essere ripercorse più volte perchè quando si sviluppano le fasi successive ci si può rendere conto di cambiamenti necessari su quelle precedenti. Questo è tipico di un approccio ad oggetti.

L'ambiente fornisce una serie di classi predefinite che possono essere utili nella modellizzazione dei processi. Alcune sono indispensabili per il funzionamento del sistema stesso, altre sono state dichiarate solamente per comodità dell'utente. Una delle classi predefinite è ad esempio quella delle attività che possiedono delle precondizioni e postcondizioni, sia statiche che dinamiche, ed un codice che va eseguito quando le precondizioni siano tutte verificate. All'attività sono anche associati uno strumento ed un ruolo che la devono sviluppare.

Il passaggio dei messaggi avviene chiamando una opportuna procedura predefinita che ha come parametri il chiamante, il chiamato, il tipo di messaggio e dei valori di ritorno che hanno semantica diversa a seconda del servizio richiesto. Si ha collegamento dinamico ('dynamic binding') del codice del metodo alla chiamata e viene utilizzato un protocollo sincrono tra trasmittente e ricevente.

Esistono anche tipi di messaggio predefiniti che permettono di leggere o scrivere attributi del chiamato, oppure conoscere gli oggetti ad esso collegati da una certa relazione.

Le attività sono organizzate gerarchicamente, cioè suddivise in sottoattività. Quando se ne esegue una e l'Execution Manager si rende conto che essa è composita, viene chiamato il Planner che istanzia i suoi figli e quindi restituisce il controllo all'Execution Manager che li esegue. Questa gerarchia di attività istanziate con le loro precedenze si dice *rete delle attività* ('*task network*').

Le relazioni di precedenza tra le attività non sono espresse esplicitamente, ma nascono dal fatto che gli oggetti in uscita da una siano presi in ingresso da un'altra. È il Planner stesso che si rende conto delle relazioni di precedenza esistenti e così nell'istanziare le attività crea la 'task network'.

Anche il problema delle modifiche alla gerarchia delle classi è stato affrontato: quando si richiede di fare un cambiamento ad una certa classe il PM Manager controlla le implicazioni di tale operazione su tutte le sottoclassi e le rispettive istanze delle classi coinvolte. Se vengono rilevati particolari problemi, quali inconsistenza o non rispetto di assunzioni generali, la modifica viene proibita. Se non si riscontrano problemi, il cambiamento è confermato convertendo automaticamente le sottoclassi e le varie istanze.

Se invece l'operazione ha conseguenze accettabili, il PM Manager crea una biforcazione dell'albero delle classi a partire dalla classe coinvolta nel cambiamento. Un ramo mantiene le sottoclassi invariate e l'altro quelle modificate. Quindi considera ad una ad una le varie istanze e, se possibile, le converte secondo le modifiche oppure no. Come si può facilmente capire una tale procedura può rendere istanze di classi diverse oggetti che prima appartenevano alla stessa, per cui si deve cercare di capire se una tale operazione sia conveniente oppure da evitare.

4.7 Conclusioni

Molti dei linguaggi utilizzati dagli ambienti per la modellizzazione dei processi permettono la definizione di tipi di dato astratti, cioè di tipi di dato con una certa struttura e ben determinati operandi che agiscono su di essi (da inizializzazione a manipolazione). È questo il concetto di classe tipico della metodologia ad oggetti.

Molto sfruttato è l'uso di tecniche di analisi e di progettazione diffuse nel campo della produzione del software, per rappresentare modelli di processi ('design and analysis paradigm' [Mad91]).

Altro approccio ora in voga è quello comportamentale in cui si descrivono le attività necessarie allo sviluppo di un prodotto con particolare interesse agli effetti di queste attività e non tanto alla loro implementazione ('behavioral approach').

Diffuso è anche l'utilizzo di insiemi di mappe ordinati gerarchicamente in cui sono descritti i metodi per raggiungere certi obiettivi ed i vari obiettivi di alto livello sono a loro volta suddivisi in insiemi di sottoobiettivi che si cerca di soddisfare mediante determinate azioni.

Ovviamente in tutti questi ambienti e linguaggi, la necessità principale è quella di descrivere e possibilmente attuare i modelli dei processi. Tutte le realizzazioni considerate hanno problemi con la modellizzazione degli strumenti automatici. Anche questi ultimi si possono vedere come istanze di un certo tipo che, come tutte le altre, sono soggette ad evoluzione per cui si possono avere versioni diverse (ad esempio di un compilatore). Così diviene necessario gestire anche questo aspetto permettendo di scegliere non solo lo strumento più adatto in un certo momento, ma anche la sua versione maggiormente calzante al problema in esame.

Bisogna comunque tener presente che quella della modellizzazione dei

processi è un'area di studio ancora giovane come si può facilmente notare dall'alta ambiguità semantica presente nei linguaggi già disponibili e dalla mancanza di una rappresentazione consolidata e universalmente riconosciuta.

Inoltre sono presenti notevoli lacune e questioni ancora aperte soprattutto in alcuni campi:

- rappresentazione delle interazioni tra le persone;
- meccanismi per gestire errori ed eventi imprevisti;
- meccanismi per la modifica del processo durante la sua esecuzione;
- integrazione tra modellizzazione e gestione dei processi, cioè tra pianificazione delle varie attività e controllo della loro esecuzione;
- integrazione tra modellizzazione dei processi e dei dati, cioè le problematiche legate alla gestione delle basi di dati e alle transazioni di lunga durata.

A dimostrare la scarsa potenza espressiva dei linguaggi di modellizzazione dei processi attualmente disponibili, ci sono i casi di studio proposti dall'International Software Process Workshop (IPSW) che danno così anche lo spunto per nuove modifiche ed estensioni a questi strumenti.

Inoltre molto lavoro deve essere ancora fatto nell'esprimere in modo definitivo le caratteristiche dei linguaggi usati che essendo in via di evoluzione presentano sintassi e semantica non ben definiti [ABGM92].

Nella tabella 4.1 sono riportati i linguaggi di descrizione utilizzati dai principali sistemi per la modellizzazione dei processi.

Sistema di modellizzazione dei processi	**Formalismo utilizzato**
Adele	modello Entità/Relazioni regole Evento-Condizione-Azione
ALF	modello Entità/Relazioni/Attributi regole
APPL/A	estensione di Ada (procedurale e basato su regole)
IPSE 2.5	linguaggio di programmazione concorrente ed imperativo
DesignNet	Reti di Petri con grafi AND/OR rappresentazione ad oggetti delle entità coinvolte nel processo e delle loro relazioni
EPM	automi a stati finiti
Epos	Orientamento agli Oggetti ed Entità/Relazioni classi rappresentate esplicitamente (riflessività)
HFSP	grammatica ad attributi
ISTAR	contratto dinamico
MARVEL	tipi per la rappresentazione dei dati regole
MELMAC/MSP	a basso livello: Reti di Petri modificate (FUNSOFT net) ad alto livello: tipi di oggetti, attività e processi
MERLIN	regole di tipo Prolog
MVP-L	linguaggio testuale tipi per i modelli generici particolare enfasi sul riutilizzo
Oikos	programmazione logica paradigma delle 'blackboard' con agenti
OPM	programmazione dei processi in Galois linguaggio ad oggetti (estensione di C++)
SPADE	linguaggio SLANG basato su Reti di Petri ad alto livello (reti ER)
SPECIMEN	reti FUNSOFT regole MERLIN

Tabella 4.1. Formalismi per la modellizzazione dei processi software

Parte II

L'orientamento agli oggetti: concetti e strumenti

5

Caratteristiche generali dell'orientamento agli oggetti

Per poter applicare l'orientamento agli oggetti alla modellizzazione dei processi software, è necessario avere una panoramica dei concetti fondamentali del paradigma ad oggetti in modo da poterli raffrontare con le necessità e con gli obiettivi del PM. In questo capitolo sono anche elencati i principi fondamentali su cui si basa il paradigma ad oggetti, perché la loro conoscenza permette di apprezzare meglio la potenza dell'orientamento agli oggetti e di sfruttarne pienamente gli strumenti più efficaci.

5.1 Concetti e terminologia

Prima di poter parlare degli aspetti che caratterizzano l'orientamento agli oggetti, è necessario definire alcuni concetti su cui esso si basa in modo da disporre di una terminologia che consenta di trattare l'argomento con chiarezza.

- Oggetto (anche detto istanza): ha uno stato, un comportamento, e un'identitá: la struttura e il comportamento di oggetti simili sono definiti in una classe.
- Classe: un insieme di oggetti che condividono struttura e comportamento comune.
- Meta-classe: la classe di una classe, cioè una classe le cui istanze sono classi.

- Relazione (anche detta associazione): esprime un legame tra oggetti definendo che sono in qualche modo correlati. Questa definzione è abbastanza generale da comprendere le relazioni classe_classe, nel caso in cui una relazione esista tra istanze di di meta-classi.

  La molteplicità di una relazione definisce quante istanze di una classe si possono collegare a un'istanza della classe associata.

Relazioni tra classi

Si possono individuare le seguenti categorie di relazioni tra classi [Boo91], che sono di particolare rilievo nell'orientamento agli oggetti.

1. Relazione di ereditarietà ('*inheritance*') definisce una relazione tra classi tale che una classe condivide la struttura e il comportamento definito da una o piu' classi (rispettivamente ereditarietà singola e multipla). Tipicamente una sottoclasse estende o ridefinisce la struttura e il comportamento delle sue super-classi.

2. Relazione di *aggregazione*: è la relazione *intero/parte* ('*whole/part*' o '*part of*') secondo la quale gli oggetti che rappresentano una parte di qualcosa sono associati a un oggetto che rappresenta il tutto.

3. Relazione di *uso*: defisce una relazione tra classi tale che una classe usa un'altra classe.

5.2 Aspetti principali e raffronti con la Modellizzazione dei Processi

Un approccio orientato agli oggetti aiuta ad affrontare la complessità intrinseca a differenti tipi di sistemi. La metodologia ad oggetti non rompe definitivamente con ciò che veniva fatto in passato, ma costruisce sulle basi ormai consolidate e sicure dei metodi precedenti.

Vediamo innanzitutto di dare un significato più preciso al concetto di *oggetto* diffusamente utilizzato in questo ambito. Gli oggetti possono essere rappresentazione di entità effettivamente tangibili e presenti nel dominio in analisi, ma possono anche essere entità astratte utilizzate per modellizare il problema che si sta trattando.

Dal punto di vista puramente informatico, gli oggetti sono "entità che combinano le proprietà di procedure e dati dal momento che eseguono elaborazioni e memorizzano uno stato locale" [SB86].

Gli oggetti hanno la caratteristica di avere un comportamento ben definito e di non poter essere utilizzati in modo diverso da quello per cui sono stati creati. Infatti essi sfruttano regole di visibilità per cui non è possibile dall'esterno modificare il loro stato, se non utilizzando le procedure che essi mettono a disposizione e che compiono l'operazione in un modo ben preciso e sicuro.

Bisogna tener presente che ogni oggetto è istanza di un tipo di dato astratto che è detto *classe*. Le diverse classi sono organizzate in una struttura ben precisa, determinata dalle relazioni che intercorrono tra di esse ed i loro oggetti, che fornisce al modello del dominio in analisi una notevole solidità.

Le classi, così come le loro istanze oggetti, contengono un certo numero di *attributi* e *metodi*. Questi ultimi permettono ad altri oggetti di agire sugli attributi i cui valori definiscono lo stato di ogni singolo oggetto. Caratteristica molto importante delle classi è l'*ereditarietà* degli attributi e dei metodi che le compongono. Cioè la sottoclasse di una certa classe possiede tutti gli attributi e metodi della sua superclasse (ed eventualmente altri in più).

Spesso per la modellizzazione dei processi si manifesta vantaggiosa una metodologia *oggetto–relazione* ('*object–relation*'). Cioè gli oggetti sono legati da relazioni (più elaborate di quelle standard della metodologia ad oggetti classica) che sono anch'esse istanze di classi e che servono a specificare nel modello attuabile quali oggetti sono connessi ad altri ed in che modo. Nelle implementazioni di queste relazioni in genere si usano istanze di classi opportunamente definite perché non sono diffusi linguaggi di tipo oggetto–relazione, nonostante ciò il concetto di relazione é ben differente da quello di oggetto.

L'approccio orientato agli oggetti porta ad una decomposizione del modello in un insieme di oggetti, tutti correlati tra loro. Ci sono comunque diverse scuole e metodologie sia di analisi che di progetto ad oggetti. Infatti l'uso della programmazione e progettazione ad oggetti, ha influenzato anche il campo dell'analisi. Le tecniche di analisi ad oggetti esaminano le entità presenti nella realtà del dominio del problema e individuano le rispettive classi ed oggetti nel dominio della soluzione.

L'approccio ad oggetti deriva, in parte, dal modello evolutivo del ciclo di vita del software [GFM+91]. Questo propone di operare mediante un certo

numero di passi ognuno dei quali dà un contributo al prodotto finale, ma soprattutto ogni passo utilizza l'incremento prodotto dal passo precedente per meglio fornire il proprio contributo. Questi concetti possono essere applicati allo sviluppo del software passando attraverso fasi di analisi, progettazione e programmazione ad oggetti. Le varie fasi non sono strettamente sequenziali, ma l'esecuzione di ognuna di esse può fornire informazioni valide per raffinare il prodotto delle altre. In particolare l'analisi e la progettazione ad oggetti sono strettamente legate.

L'approccio incrementale non caratterizza solamente analisi, progettazione ed implementazione, ma anche le attività che le costituiscono. Ad esempio nella progettazione ad oggetti si possono individuare delle attività, ognuna delle quali permette di realizzare una parte o un'aspetto del prodotto finale (progetto). Anche queste attività sono affrontate con un approccio incrementale ed iterativo realizzando quello che in letteratura è detto '*round-trip gestalt design*'.

5.3 Principi fondamentali

L'orientamento ad oggetti comprende sia metodologie di analisi, sia di progettazione, ed anche linguaggi di programmazione. In questa parte sull'orientamento agli oggetti vengono trattati prevalentemente gli aspetti legati all'analisi ed alla progettazione.

La metodologia Coad/Yourdon considera come analisi tutta la parte di modellizzazione indipendente dall'implementazione del componente del dominio del problema; invece nella metodologia del Booch la modellizzazione è completamente situata nell'ambito della progettazione. Secondo la metodologia Coad/Yourdon il passaggio alla fase di progettazione avviene quando si specificano più profondamente, eventualmente aggiungendo dettagli implementativi, il componente del dominio del problema e gli altri tre. Tutto questo è un'ulteriore conferma del fatto che il confine tra analisi e progettazione non è ben marcato ed univocamente individuato, tanto più in un approccio incrementale come quello ad oggetti.

Nel seguito di questa sezione sono descritti i principi su cui si basano analisi e progettazione secondo Booch [Boo91] e secondo Coad e Yourdon [CY91a].

Astrazione

È uno dei metodi di cui l'uomo dispone per affrontare la complessità. L'uso dell'astrazione permette di evidenziare le similitudini (ad esempio tra certi oggetti o certe classi) e gli aspetti più salienti di un certo problema, trascurando gli altri aspetti che sono di secondo piano.

Questo è di grande aiuto e supplisce alle limitate capacità umane perché, focalizzando l'attenzione solo sugli aspetti principali è più facile creare una visione di insieme del problema trattato per comprenderlo appieno.

L'astrazione è poi particolarmante efficace in un approccio ad oggetti, perché consente di individuare le caratteristiche di un oggetto che lo distinguono dagli altri, definendone i confini. Questa delimitazione dell'oggetto è però soggettiva, cioè dipende dal punto di vista di chi sta analizzando il dominio.

Allora l'astrazione fornisce una visione esterna degli oggetti, ma è indispensabile perché definire il giusto insieme di astrazioni per rappresentare un certo dominio, è il problema principale della progettazione orientata agli oggetti.

Infine, grazie all'astrazione, non solo si identificano le entità coinvolte in una certa reltà, ma anche i legami tra queste, cosa mettono a disposizione, le operazioni che eseguono le une sulle altre e gli scopi con cui le fanno.

Associazione

Consiste nella capacità di vedere la connessione esistente tra cose o eventi che si manifestano in modo analogo. Questa capacità è molto importante nella progettazione ad oggetti per il fatto che permette di individura le relazioni tra i vari oggetti che sono indispensabili per dare una strutturazione completa ed efficace ai modelli che si realizzano.

Incapsulamento

Come accennato in precedenza gli oggetti hanno un comportamento ben preciso e l'implementazione di tale comportamento è completamente interna all'oggetto e non accessibile dal mondo esterno. Questo ha un duplice vantaggio che sta nel fatto che il loro comportamento è definito una volta per tutte ed è indipendente da chi ne vuole fare uso, cioè dal *cliente*.

Inoltre cambiando l'implementazione di tale comportamento, se le sue

specifiche continuano ad essere rispettate, non si crea alcun problema ai clienti. Infatti le interazioni di un oggetto con l'esterno ed i servizi che questo fornisce, sono indipendenti dalla sua implementazione, ma legati sostanzialmente alla sua *interfaccia*. Così cambiamenti dell'implementazione che non coinvolgano anche l'interfaccia, non creano problemi al mondo esterno.

È bene notare che astrazione ed incapsulamento sono concetti strettamente legati e tra loro complementari. Infatti, per poter inizialmente trascurare i dettagli implementativi focalizzando la propria attenzione su quelli che sono gli aspetti più rilevanti del problema, si deve avere la garanzia di poter trattare in un secondo tempo i dettagli implementativi senza coinvolgere i punti fermi ormai definiti.

Da questo concetto deriva la necessità di organizzare le classi in interfaccia ed implementazione con ben determinate regole di visibilità. L'interfaccia è visibile dall'esterno e regola i rapporti e le interazioni con altri oggetti appartenenti alla stessa classe o ad altre, stabilendo i servizi forniti dagli oggetti della classe in questione. Invece l'implementazione non è visibile dall'esterno, cioè da parte di altri oggetti ed è ciò che fa sì che il comportamento dell'oggetto sia quello voluto.

Modularità

Questo è un concetto prettamente legato alla struttura fisica dei prodotti software, però come già detto nei capitoli precedenti riveste una notevole importanza anche nel campo dei modelli di processo.

La divisione di prodotti software in moduli garantisce una maggiore facilità di comprensione del funzionamento del prodotto stesso, una maggiore maneggevolezza ed uno strumento per aggirare i problemi dovuti alle limitazioni sulle dimensioni. In ogni caso la modularità è un concetto indispensabile per la progettazione ad oggetti, perché la suddivisione in moduli di un prodotto va tenuta presente fin dalla fase di progetto.

Considerando i modelli di processo, come detto più volte, questi possono essere visti come qualunque prodotto software e quindi trattati nello stesso modo (analisi, progetto, codifica, ecc.) applicando anche ad essi il concetto di modularità che si esplicita nel raggruppare le varie attività in moduli. Questo aspetto è comunque secondario per la modellizzazione dei processi dal momento che è legato alla visione fisica del sistema.

Gerarchia

L'astrazione è un punto chiave della progettazione ad oggetti. Però spesso non basta un solo livello di astrazione per avere una chiara comprensione di un problema. Allora si rivela necessaria una gerarchia di astrazioni per meglio modellizzare il dominio in esame.

Si possono individuare due tipi principali di gerarchia:

Tipo di ('kind of') Si ha tra due astrazioni quando una è una generalizzazione dell'altra; infatti si dice che la gerarchia 'kind of' stabilisce tra due classi una relazione del tipo *specializzazione/generalizzazione* ('*gen/spec*').

Questa gerarchia è molto importante perché una sua espressione è costituita dall'*ereditarietà* ('*hineritance*') che è uno dei concetti di base della progettazione e dei linguaggi di programmazione ad oggetti. Cioè una classe può avere delle sottoclassi che ne ereditano tutte le caratteristiche (attributi e comportamento) aggiungendone eventualmente altri.

Si possono avere casi in cui è utile che una classe erediti le caratteristiche di più di una classe. Questo è possibile qualora il formalismo o il linguaggio ad oggetti utilizzato supportino l'*ereditarietà multipla* ('*multiple hineritance*'). Questo meccanismo, sebbene molto utile e potente, può talvolta portare a situazioni critiche: ad esempio avendo lo stesso attributo o metodo dichiarato in due delle superclassi di una classe.

Si dice che la gerarchia 'kind of' stabilisce tra due classi una relazione del tipo *specializzazione/generalizzazione* ('*gen/spec*').

Parte di ('part of') Questa indica che un'astrazione è costituita dall'unione di differenti astrazioni. In una gerarchia di classi tipica di un approccio ad oggetti una struttura di questo tipo definisce una relazione di *aggregazione* tra un insieme di classi ed un'altra classe, dtsndo ad indicare che ogni oggetto di quest'ultima è ottenuto come aggregato di un certo numero di istanze delle altre. Questa relazione è detta anche *intero/parte* ('*Whole/Part*').

Comunicazione mediante messaggi

Le persone comunicano tra loro mediante lo scambio di messaggi, siano essi scritti o verbali. Si può allora pensare di definire meccanismi di comunicazione analoghi tra gli oggetti creando delle opportune interfacce di comunicazione che definiscono quali siano i messaggi ammissibili.

Un tale approccio fornisce notevole solidità e sicurezza grazie alla definizione delle interfacce, ma soprattutto ha il vantaggio di essere familiare al progettista dal momento che si tratta di un meccanismo di comunicazione analogo a quello utilizzato dalle persone.

Tipizzazione

Questo concetto è fortemente legato alla programmazione, sia essa orientata agli oggetti o sia essa di altro genere. In un contesto ad oggetti la tipizzazione si esprime nel fatto che ogni classe rappresenta un tipo che non può essere in alcun modo scambiato od utilizzato al posto di un altro.

I diversi linguaggi presentano livelli diversi di rigidità nell'uso della tipizzazione; una notevole rigidità è utile soprattutto quando si sviluppino prodotti di grosse dimensioni per scoprire facilmente errori che possono essere rilevati già durante la compilazione. Ovviamente una rigida tipizzazione diminuisce la flessibilità del linguaggio rendendo più difficili determinati tipi di genericità e riutilizzo.

Ovviamente questa caratteristica può essere importante anche per applicazioni più generali della semplice programmazione come può essere la modellizzazione dei processi.

Legato alla tipizzazione è il collegamento dinamico o statico ('dynamic or static binding'). Il primo fornisce maggiore flessibilità perché permette di assegnare ad un nome un tipo (ovvero un oggetto ad una determinata classe) solo al momento dell'esecuzione. Questo si rivela così uno strumento molto potente in mano al progettista ed al programmatore che va sotto il nome di *polimorfismo*. Ovviamente tale meccanismo genera una maggiore vulnerabilità agli errori in analogia a quanto detto sopra a riguardo della tipizzazione più o meno rigida.

Concorrenza

È questo un altro aspetto prettamente legato alla programmazione, ma che si riflette anche sulla progettazione. Non è comunque un aspetto di secondo piano nella modellizzazione dei processi dal momento che, come già più volte evidenziato, i processi software sono intrinsecamente concorrenti.

Nella metodologia ad oggetti la concorrenza è esplicitata nel fatto che ogni oggetto può essere pensato come un'entità che ha una vita indipendente da quella delle altre ed ha quindi un suo sviluppo dell'esecuzione ('*thread*

of control'). Questo è molto utile nella modellizzazione dei processi perché, ad esempio il modello di due programmatori che scrivono codice, non può che tenere in conto che le due azioni avvengono contemporameamente ed in modo assolutamente indipendente.

Gli oggetti che posseggono questo 'thread of control' sono detti *attivi*. Il mondo in una visione orientata agli oggetti può essere visto come un insieme di oggetti che interagiscono tra loro. Alcuni eseguono determinate operazioni e servizi solo quando richiesti e la loro risposta è attesa dal cliente (non attivi). Altri hanno un loro comportamento ed evoluzione indipendente quando agiscono su o interagiscono con altri (attivi).

Persistenza

Nella programmazione ci sono oggetti che hanno vita di durata diversa che può essere inferiore o uguale a quella della procedura che li ha creati, oppure superarla.

Nella modellizzazione dei processi quasi tutti gli oggetti hanno una vita lunga anche perché i processi stessi hanno notevole durata. Quindi tali oggetti devono spesso sopravvivere alle attività che li hanno creati. Inoltre la loro vita comprende anche una certa evoluzione per cui la gestione della persistenza nell'ambito della modellizzazione dei processi è molto più complessa che quella delle strutture dati create nei programmi.

Si rivelano quindi necessarie tecniche particolari ed eventualmente l'appoggiarsi a strutture già esistenti e ben funzionanti quali possono essere basi dati ad oggetti ed efficienti gestori delle stesse. Queste basi dati sono costruite facendo uso di tecnologie consolidate (modelli sequenziali, indirizzati, gerarchici, relazionali), ma forniscono un'interfaccia orientata agli oggetti con cui si può facilmente gestire la persistenza–ed eventualmente l'evoluzione–degli oggetti di cui è costituito il modello.

In sistemi distribuiti, come può essere un ambiente per la modellizzazione dei processi e la loro esecuzione, è importante la persistenza, non solo nel tempo, ma anche nello spazio, cioè la capacità di far migrare ed eventualmente condividere i vari oggetti.

5.4 Vantaggi del paradigma ad oggetti

L'utilizzo di metodologie orientate agli oggetti, sia in ambito di analisi, che di progettazione, che di programmazione–intesa in senso lato–, offre numerosi vantaggi.

- *Riutilizzo*: è caratteristico del modello ad oggetti. Questo consiste nel riutilizzare non solo parti di codice, ma anche di progetto o di analisi. Il riutilizzo è uno dei punti di forza del modello ad oggetti.

- *Diversi livelli di astrazione e di gerarchie* di oggetti che si basano gli uni sugli altri. Questo perché se gli strati inferiori, su cui si basano quelli superiori, sono stati sviluppati correttamente e sono ben consolidati, la struttura globale risulta affidabile.

- *Attitudine e robustezza ai cambiamenti* derivano dall'uso dell'incapsulamento. Infatti cambiare l'implementazione di un oggetto non richiede di cambiare tutti gli oggetti che hanno con esso una qualche relazione.

- *Sicurezza*: se la struttura degli oggetti è stata oculatamente definita, l'incapsulamento impedisce di perturbare il corretto funzionamento dell'oggetto agendo dall'esterno. Ciò ha come conseguenza che l'uso degli oggetti può essere fatto senza troppe precauzioni perché il loro corretto comportamento è stato pienamente definito nell'implementazione.

- *Procedimento incrementale* [GFM+91]. Questo approccio ha una un processo di sviluppo differente rispetto a quelli utilizzati dalla maggior parte delle metodologie precedenti (ad esempio il modello a cascata) che non richiede di separare le varie fasi dello sviluppo in modo netto cominciando quelle successive solo quando le precedenti siano completamente terminate.

  In altre parole l'approccio orientato agli oggetti consente di operare per raffinamenti successivi. Ciò significa che non è necessario finire l'analisi prima di cominciare il progetto e finire quest'ultimo prima di cominciare a codificare, ma si può passare dall'analisi alla progettazione ed alla codifica, per poi tornare indietro a raffinare l'analisi e così via.

  Così facendo è necessario un minor sforzo intellettuale e si acquisisce una maggiore *conoscenza* del dominio in modo *graduale* avendo poi

la possibilità di applicarla alle varie fasi del processo. Se invece non si contemplasse la possibilità di tornare sui propri passi, la padronanza del dominio in esame acquistata andrebbe sprecata o comunque utilizzata solo nelle fasi ancora da sviluppare.

- *Familiarità*: un approccio orientato agli oggetti si rivela vantaggioso perché i concetti del modello ad oggetti sono molto *familiari* al modo di operare della mente umana e quindi facilmente assimilabili ed utilizzabili in modo fruttuoso [Rob81]. Come già introdotto nella sezione 5.3, i principi su cui si basano le metodologie ad oggetti sono gli stessi che l'uomo utilizza per affrontare la complessità.

L'applicazione del paradigma ad oggetti alla Modellizzazione dei Processi aggiunge a quelli sopraelencati altri vantaggi. Il più evidente è che con gli strumenti propri dell'orientamento agli oggetti si possono facilmente rappresentare entità astratte (meta-processi, processi software, modelli di processo) e reali (strumenti, persone, risorse, ecc.) tipici dell'ambito della modellizzazione dei processi.

6

Metodologia Coad/Yourdon di progettazione ad oggetti

Si è voluta utilizzare la metodologia Coad/Yourdon per la progettazione di un ambiente PM ed anche per la realizzazione dei modelli di processi da simulare in questo ambiente. Allora è bene conoscere gli apetti che caratterizzano questa metodologia; in questo capitolo è descritta la struttura che, secondo gli autori, deve avere un progetto ad oggetti e quindi le attività di progettazione che derivano dall'imposizione di una tale struttura. Infine sono riportate alcune considerazioni sulla adeguatezza della metodologia Coad/Yourdon alla progettazione di S^3 ed alla modellizzazione dei processi software, illustrando le convenzioni che si sono adottate nell'utilizzo della metodologia in questione per gli scopi di questo lavoro.

6.1 Caratteristiche ed aspetti principali

La metodologia Coad/Yourdon pone particolare enfasi sull'analisi del problema più che sulla sua progettazione. L'analisi, indipendentemente dal tipo di approccio utilizzato, ha lo scopo di descrivere le caratteristiche del dominio del problema, cioè *cosa* il sistema deve offrire. La progettazione descrive *in che modo* il sistema può essere realizzato. Allora l'analista ha il compito di investigare il dominio del problema ed individuare quali siano le responsabilità del sistema in tale dominio, restando ad un livello indipendente dall'implementazione e dai suoi dettagli. La progettazione è un'espansione dell'analisi e fornisce una particolare implementazione [CY91a, pag. 178].

La metodologia Coad/Yourdon dà notevole importanza alla possibilità del riutilizzo, caratteristica tipica degli approcci ad oggetti, cercando di riutilizzare addirittura i risultati dell'analisi.

La notazione proposta non fornisce strumenti per rappresentare il livello delle istanze, se non nelle descrizioni delle varie classi. Questa notazione sarà trattata nella sezione A.3, in cui si fa riferimento alla simbologia utilizzata dallo strumento automatico di supporto alla progettazione di cui ci si è serviti per modellizzare l'esempio di processo software proposto nella sezione 11.3.

6.2 Componenti del modello del progetto

Il modello del progetto è costituito da quattro componenti. Le quattro principali attività della progettazione constano nel progettare questi quattro componenti [CY91b].

Componente dell'interazione con le persone Include le interfacce che sono necessarie per l'interazione tra la macchina e l'uomo, e tipicamente classi per la gestione di finestre.

Componente del dominio del problema Contiene i risultati dell'analisi, cioè le classi e gli oggetti che sono stati individuati per rappresentare il problema in esame. Le classi e gli oggetti di questo componente possono non coincidere con quelli ricavati mediante l'analisi, perché modificate al subentrare di vincoli di progettazione come ragioni di tempistica, dimensioni, divisione in blocchi per la memorizzazione, ecc.

Componente per la gestione dei 'task' Si occupa della definizione dei processi necessari, della loro comunicazione e del loro coordinamento. Inoltre in questo componente è possibile inserire informazioni relative all'allocazione fisica dei dispositivi ed ai protocolli con sistemi e dispositivi esterni.

Componente per la gestione dei dati Specifica le modalità di accesso ai dati e la gestione della loro persistenza separando dal resto del modello gli aspetti che riguardano l'uso di basi di dati o di semplici 'file'.

La metodologia Coad/Yourdon dà importanza alla prototipizzazione fin dalle prime fasi dell'analisi. Questa risulta essere vantaggiosa per quanto riguarda il componente dell'interazione con le persone la cui prototipizzazione può mostrare l'aspetto delle interfacce del prodotto finale.

Nella parte III, in cui è riportato il modello ad oggetti dell'esempio di processo software, si è sviluppata solamente quella che secondo la metodologia Coad/Yourdon è la fase di analisi del sistema e cioè si sono riportate solamente le entità che servono a modellizzare il componente del dominio del problema[1]. Le altre sono state prototipate direttamente in Smalltalk usando i meccanismi da esso predefiniti.

6.3 Strati del modello

In ognuno dei componenti sopraelencati si possono individuare cinque strati:

- soggetti
- classi ed oggetti
- struttura
- attributi
- servizi

Attraversando questi strati ci si immerge, a livelli di dettaglio sempre maggiore, nel modello. Uno strumento automatico per l'analisi e la progettazione ad oggetti secondo la metodologia Coad/Yourdon, deve permettere all'utente di vedere uno o più di questi strati a sua scelta.

6.4 Attività della metodologia

L'analisi si occupa della definizione delle classi e le loro relazioni, per quanto riguarda il componente del dominio del problema. Le cinque pricipali attività dell'analisi corrispondono alla definizione dei cinque strati che costituiscono il componente del dominio del problema:

[1] Nel seguito ci riferiremo a questa fase come ad una fase di progettazione.

- ricerca delle classi e degli oggetti
- identificazione delle strutture
- identificazione dei soggetti
- definizione degli attributi
- definizione dei servizi

Queste sono attività e non sono necessariamente sequenziali; l'ordine di esecuzione può essere scelto a piacere, a seconda delle preferenze del progettista, ma soprattutto si può tornare a portare avanti attività che si erano precedentemente iniziate e quindi sospese per passare ad altre. Queste attività guidano da livelli di astrazione alti, a livelli sempre più bassi; può allora essere utile ripercorrerle più volte raffinandone successivamente il prodotto con un procedimento incrementale ed iterativo tipico dell'orientamento agli oggetti, ma anche del modello evolutivo del ciclo di vita del software [GFM+91].

Nelle sezioni successive sono riportati i vari passi da seguire per portare a termine queste fasi come suggerito dalla metodologia.

La notazione propria della metodologia Coad/Yourdon verrà specificata nella parte III riguardante la modellizzazione dell'esempio di processo software con particolare riferimento a quella adottata dallo strumento automatico di sussidio alla progettazione utilizzato (sezione A.3).

6.4.1 Ricerca delle classi e degli oggetti

Le classi ed i loro oggetti vanno individuati nel dominio del problema, ma soprattutto tra le responsabilità del sistema da modellizzare all'interno di tale dominio. Per poter individuare questi elementi è necessario conoscere il dominio in esame in prima persona, oppure parlare con addetti ai lavori ed esperti nel campo, in modo da acquisire il maggior numero possibile di informazioni. È importante in ogni caso provare ad immergersi in prima persona nel dominio del problema, anche quando si stia effettuando l'analisi seguendo un approccio non ad oggetti. Può essere utile anche studiare i risultati di analisi precedenti del sistema in esame o di sistemi simili.

Fin da questa prima fase è bene realizzare prototipi che sono utili sia all'analista, che al cliente che si può meglio rendere conto di cosa si sta producendo ed eventualmente se è effettivamente riuscito a comunicare quello che desiderava all'analista.

Entità particolarmente rilevanti

Per meglio individuare le classi e gli oggetti necessari a modellizzare il sistema, nell'analizzare il dominio del problema, si devono considerare gli elementi sottoelencati.

- Le *strutture*, siano esse di tipo generalizzazione/specializzazione o intero/parte, che compaiono nel dominio del problema. Queste sono rispettivamente un modello della gerarchia 'kind of' e 'part of' descritte nella sezione 5.3 per la metodologia Booch.
- Gli *altri sistemi* con cui il sistema in esame interagisce, comprese le persone.
- I *dispositivi* con cui interagisce e di cui ha bisogno. Ciò di cui effettivamente si deve tenere conto in questa fase, è la rappresentazione astratta di questi dispositivi da cui il sistema riceve informazioni o di cui ne mantiene. È bene non considerare dispositivi dipendenti dall'implementazione, ma demandare qualunque considerazione su di essi alle fasi successive della progettazione.
- Le entità o gli *eventi* di cui il sistema deve mantenere traccia.
- I *ruoli* delle persone che interagiscono con il sistema, o sui quali il sistema mantiene informazioni.
- Le *procedure di operazione* che descrivono le azioni svolte dal sistema come, per esempio, l'interazione con le persone o particolari ordini di attuazione.
- I *luoghi* in cui possono trovarsi elementi di rilievo del dominio del problema (ad esempio l'ufficio di una certa persona).
- L'*organizzazione* e la suddivisione del sistema e delle sue risorse, comprese quelle umane.

Spunti per operare una prima selezione

Una volta individuato un certo numero di classi e loro oggetti è necessaria una valutazione del loro effettivo contributo al modello, e cioè se siano effettivamente indispensabili alla modellizzazione del sistema. Per meglio svolgere

questo compito, per ciascuna delle entità in esame si possono analizzare gli aspetti riportati nel seguito.

- Considerare se il sistema richieda di mantenere informazioni su un certo oggetto e di che tipo queste debbano essere. A questo livello è possibile anche individuare alcuni degli *attributi*, cioè che cosa vada memorizzato.
- Valutare se il sistema richieda ad un certo oggetto dei servizi ed in caso affermativo stimarne la natura in modo da definire, già a questo livello, alcuni dei *metodi* della classe corrispondente (per lo meno quelli di base).
- Porre particolare attenzione agli oggetti che hanno un singolo attributo, perché spesso situazioni di questo genere possono essere un campanello di allarme che indica che l'oggetto in questione non è strettamente necessario all'interno del sistema, ma ciò che rappresenta può essere modellizzato diversamente.
- Sospette sono anche quelle classi con un solo oggetto, che possono spesso rivelarsi superflue ad una più attenta analisi.
- Quando ci si trova di fronte ad attributi o servizi che non sono utili per qualunque istanza della classe, è senza dubbio meglio introdurre delle strutture di generalizzazione/specializzazione.
- Curarsi dei requisiti che il sistema deve avere, cioè non solamente di quelli richiesti dal cliente che possono spesso essere incompleti. Inoltre non ci si deve preoccupare dei requisiti che derivano o sono influenzati da una particolare implementazione, perché essi vanno demandati alle fasi successive della progettazione.
- Si deve cercare di evitare di mantenere informazioni ridondanti, cioè tali che possano essere ricavate da altre che sia indispensabile memorizzare. Se il mantenere tali informazioni può recare vantaggio dal punto di vista delle prestazioni, è un fatto che andrà valutato solo nelle fasi successive del progetto (si tratta infatti di aspetti tipici dell'implementazione).

6.4.2 Identificazione delle strutture

Le strutture sono di due tipi:

- **generalizzazione/specializzazione** (`Gen/Spec`): rappresenta la gerarchia di tipo 'kind of' tra due classi (sezione 5.3) e si tratta appunto di una relazione a livello classe, come accennato nella sezione 5.1.

- **intero/parte** (`Whole/Part`): è una relazione a livello istanza che indica che ogni istanza di una delle classi connesse (`whole`) è costituita da un certo numero di istanze dell'altra classe (`part`); è associata una cardinalità che esprime tale numero e di quanti interi è componente un certo oggetto parte. Questa struttura rappresenta la gerarchia di tipo 'part of' di cui si è parlato nella sezione 5.3.

Verifiche sulle strutture `Gen/Spec`

Una volta individuato un certo insieme di classi e la struttura di tipo `Gen/Spec` che le correla, si deve riesaminare il tutto per controllare che le varie classi siano effettivamente utili e le specializzazioni siano appropriate.

- Per ogni classe generalizzazione si deve controllare se le sue specializzazioni:

  - sono nel dominio del problema;
  - rientrano nelle responsabilità del sistema;
  - hanno effettivamente necessità di ereditare le caratteristiche della classe `gen`;
  - soddisfano le varie condizioni per classi ed oggetti espresse nella sezione 6.4.1.

- Analogamente per ogni classe specializzazione si deve controllare se le sue generalizzazioni rispettano i punti sopraelencati.

- Per le varie classi individuate si deve vedere se è possible stabilire generalizzazioni comuni che andranno ovviamente anch'esse sottoposte alle verifiche di cui sopra.

- È bene evitare di introdurre una classe specializzazione solamente per condividere attributi comuni alle due classi. Infatti prima di tutto è importante la comprensibilità del modello, per cui non si devono creare classi generalizzazione che non abbiano riscontro nel dominio in analisi.

Nel definire le relazioni `Gen/Spec` tra le classi si deve tenere presente che si possono creare anche strutture a grafo ('lattice') che denotano l'uso di ereditarietà multipla.

Come individuare le strutture tt Whole/Part

Le strutture di tipo intero/parte si possono individuare cercando di evidenziare tra gli oggetti presi in considerazione relazioni di tipo:

- assemblato/parte;
- contenitore/contenuto;
- unione/membro.

Nella descrizione dell'oggetto `whole` possono essere specificate eventuali caratteristiche particolari della relazione, come ad esempio il fatto che le parti costituiscano un insieme ordinato.

Verifiche sulle strutture `Whole/Part`

Anche per le classi correlate da strutture di tipo intero/parte è bene controllare che gli elementi che vi partecipano siano effettivamente utili alla modellizzazione del problema. Questo può essere fatto con passi analoghi a quelli elencati per le strutture `Gen/Spec` chiedendosi per ogni classe:

- se sia effettivamente nel dominio in esame;
- se faccia parte delle responsabilità del sistema;
- se svolge effettivamente il suo compito in modo più efficiente di quanto possa essere fatto da un singolo attributo;
- se fornisca un'astrazione utile nell'affrontare il problema in esame.

6.4.3 Identificazione dei soggetti

Il numero delle classi presenti in un modello può essere molto elevato per cui una notazione piatta causerebbe difficoltà di lettura. Occorre allora avere a disposizione uno strumento per guidare il lettore attraverso il modello.

Questa funzione è assolta dai soggetti che costituiscono uno strumento per il *controllo della visibilità* rendendo più comprensibile il problema. Un soggetto raggruppa al suo interno un certo numero di classi, ed aventualmente altri soggetti, creando una visione gerarchica del modello.

Si ottengono così differenti *livelli* annidati di diagrammi tra cui deve essere guidata l'attenzione del lettore. Particolare importanza deve essere data a questo aspetto dagli strumenti automatici di supporto ad analisi e progettazione orientati agli oggetti.

Come individuare e raffinare i soggetti

Un primo livello di soggetti si può individuare semplicemente raggruppando le classi che discendono mediante ereditarietà da un singolo padre, detto *radice*. Il nome dato al soggetto così ottenuto sarà proprio quello della classe radice. Si può arricchire l'insieme dei soggetti passando a considerare gli eventuali sottodomini del problema.

Infine è bene raffinare la divisione in soggetti tenendo presente che tra i soggetti deve esserci il minor numero possibile di:

- *dipendenze*, cioè riferimenti degli oggetti delle classi contenute nell'uno a quelli delle classi contenute nell'altro (`Instance Connection` introdotte nella sezione 6.4.4);
- *interazioni*, cioè scambio di messaggi tra gli oggetti delle classi contenute nell'uno e quelli delle classi contenute nell'altro (`Message Connection` introdotte nella sezione 6.4.5).

Quindi è bene che le classi che sono tra loro correlate vengano incluse nello stesso soggetto, così che i soggetti appaiano il più possibile indipendenti. Va tenuto presente che una stessa classe può appartenere anche a più di un soggetto.

Se il modello è molto piccolo i soggetti possono anche non essere introdotti. Quando si intende utilizzarli, si può decidere di introdurli inizialmente per evidenziare i sottodomini del problema da analizzare, oppure dopo aver già identificato un certo numero di classi per raggrupparle. Questo mette ancora una volta in risalto il fatto che le attività della metodologia non hanno un ordine predefinito ed inoltre possono essere eseguite in modo incrementale.

6.4.4 Definizione degli attributi

L'attributo è un elemento tipico del modello ad oggetti ed è definito da Coad e Yourdon come un'informazione di stato per cui ciascun oggetto in una classe ha un suo valore [CY91a, pag. 119]. Quindi gli attributi descrivono caratteristiche dell'entità modellizzata dall'oggetto che li contiene; sono utili solo gli attributi che rappresentano caratteristiche che rientrano nelle responsabilità del sistema. Sugli attributi agiscono i *servizi* (introdotti nella sezione 6.4.5) e solo mediante questi possono essere manipolati da altri oggetti.

L'attività di definizione degli attributi può essere suddivisa nelle sottoattività elencate di seguito.

Identificare gli attributi Ci si deve chiedere quali siano i compiti che un oggetto deve svolgere all'interno delle responsabilità del sistema in esame ed in quale modo. Così si può giungere all'identificazione degli *stati* attraverso cui l'oggetto deve transitare e degli attributi necessari per rappresentarli.

In questa prima fase si devono tenere presenti alcuni accorgimenti:

- utilizzare attributi atomici che hanno un singolo valore o un insieme di valori strettamente correlati;
- la normalizzazione dei dati (per evitare il più possibile la ridondanza) è bene che sia demandata alle fasi successive della progettazione;
- la decisione riguardo alla memorizzazione di un valore che può essere ricavato mediante calcolo viene demandata alla progettazione, anche perché una scelta in tale senso può essere influenzata dalla particolare implementazione;
- i meccanismi di identificazione (l'uso di chiavi, puntatori, ecc.) sono responsabilità delle fasi più avanzate della progettazione.

Per fare riferimento ad un oggetto si può considerare di avere a disposizione un identificatore implicito (che è unico) e non è riportato tra gli attributi di una certa classe. L'uso di questo identificatore è particolarmente indicato per evitare di utilizzare quelli che sono gli identificatori del mondo reale che talvolta si rivelano non essere unici, anche se in casi molto particolari.

Posizionare gli attributi È importante che gli attributi siano posizionati negli oggetti appropriati. In particolare, nelle strutture `Gen/Spec`, essi devono essere messi nella più generale classe, tale che tutte le sue sottoclassi necessitano dell'attributo in questione. In questo modo le sottoclassi disporranno di tale attributo grazie all'ereditarietà.

Specificare gli attributi Gli attrubuti vanno specificati tenendo presenti alcune regole fondamentali proprie del dominio dell'ingegneria del software, e non tipiche dell'orientamento agli oggetti o della metodologia Coad/Yourdon:

- assegnare un *nome leggibile*;
- associare una *descrizione* che riporti la funzione dell'attributo nel sistema ed in quale misura esso permette di soddisfare i requisiti;
- specificare un eventuale *valore di 'default'*;
- evidenziare gli *stati in cui è utilizzato* ed in cui il suo valore è quindi significativo;
- esprimere *vincoli di accesso o creazione*;
- specificare dei *vincoli sui valori* che esso può assumere se ciò facilita la definizione dei servizi che opereranno sull'attributo. Tali vincoli possono essere del tipo:
  - unità di misura;
  - intervallo di valori ammissibili;
  - insieme di valori assegnabili;
  - imposizione di assegnare un valore;
  - vincoli derivanti dal valore di altri attributi.

Identificare le connessioni tra istanze (`Instance Connection`) Collegano gli oggetti che necessitano di una correlazione per poter svolgere il loro compito [CY91a, pag. 127]. Come detto per gli attributi, analogamente le connessioni devono collegare le classi più generali della struttura `Gen/Spec` che necessitano di questa correlazione.

Ad ogni `Instance Connection` sono associate due coppie di interi permettono di esprimere vincoli di cardinalità. Ogni coppia definisce il numero minimo e massimo di istanze di una classe che sono connesse ad ogni istanza dell'altra, e viceversa. Se accade che, avendo più di un oggetto collegabile, ce ne sia uno con un particolare significato (ad esempio il più recente), è necessario aggiungere un attributo per tenerne conto.

Se ci sono particolari vincoli sulle istanze connesse ad un oggetto, questi possono essere espressi nella descrizione associata all'oggetto in questione.

Le strutture `Whole/Part` possono sembrare un caso particolare di `Instance Connection`, ma le prime hanno una semantica definita nell'orientamento agli oggetti ed in particolare nella metodologia Coad/Yourdon. Il concetto di intero/parte è uno dei metodi di base tipici del modo di ragionare dell'uomo; quindi il suo significato è più forte che quello di una semplice relazione tra oggetti nel dominio del problema [CY91a, pag. 128]. Inoltre il legame imposto da una connessione di istanza tra due oggetti è di carattere più generale e la semantica è data dal suo nome e da vincoli di cardinalità.

Raffinare le scelte fatte Si devono rivalutare le scelte fatte per raffinare il modello nella sua globalità, magari mettendo in discussione anche gli strati precedentemente analizzati. In questa fase è di aiuto considerare i casi particolari.

- *Attributi non utilizzati*: ci si può trovare in situazioni in cui per una classe si sia previsto un certo numero di attributi, ma che alcuni di essi non vengano effettivamente utilizzati. Questi divengono ovviamente superflui e vanno eliminati dal modello; molto più rilevante è il fatto che probabilmente va rivista anche la struttura delle classi a fronte di un tale evento, perché anch'essa può essere stata erroneamente concepita.

- *Singolo attributo*: come già espresso nella sezione 6.4.1, le classi che hanno un solo attributo possono non essere strettamente necessarie ed il loro compito nel sistema può essere svolto in modo ugualmente efficace da un'altra entità.

- *Valori ripetuti*: quando esistono attributi che assumono lo stesso valore in più di un oggetto, è probabile che l'attributo in questione possa essere sostituito da una nuova classe.

- *Connessioni tra più oggetti*: le `Instance Connection` di cardinalità n,m connettono più oggetti di una classe a più oggetti dell'altra; può essere utile introdurre una nuova classe i cui attributi permettono di identificare le diverse relazioni tra i vari oggetti.

- *Connessioni tra oggetti della stessa classe*: come nel caso precedente può essere utile introdurre una classe che identifichi e descriva la connessione.

- *Più di una connessione tra due oggetti*: si trova quando si voglia evidenziare una differenza semantica tra i due tipi di relazione. Le due connessioni possono essere evitate introducendo una classe che descriva la relazione tra gli oggetti e ne identifichi il tipo.

Controllare se sono necessarie altre connessioni Si deve prendere in considerazione ogni coppia di oggetti per valutare se tra di essi ci sia una corrispondenza nel dominio del problema che non sia in ogni momento ottenibile percorrendo `Instance Connection` già tracciate.

6.4.5 Definizione dei servizi

Un servizio definisce i modi in cui è possibile operare dall'esterno su di un oggetto e cioè le funzionalità che questo offre. Stanno alla base della comunicazione tra oggetti che avviene mediante l'invocazione dei reciproci servizi.

Un oggetto durante la sua vita attraversa degli stati che sono identificati dal valore dei suoi attributi. L'invocazione di un servizio permette di cabiare il valore degli attributi e quindi di causare il passaggio da uno stato all'altro.

L'attività di definizione dei servizi può essere divisa nelle sottoattività elencate di seguito.

Identificazione degli stati degli oggetti Ogni stato è individuato dai valori degli attributi. L'oggetto può essere caratterizzato da un diverso comportamento nei vari stati. Quindi l'identificazione di uno stato consiste in:

- individuare i valori degli attributi che lo distinguono;

- individuare i comportamenti particolari legati alla permanenza nello stato in questione.

La vita dell'oggetto attraverso i vari stati si può descrivere facendo uso dei *diagrammi di stato degli oggetti* ("*Object State Diagram*") che permettono di esprimere gli stati, i servizi che causano le transizioni e i valori degli attributi che caratterizzano lo stato (per la notazione si rimanda alla sezione A.3). Ovviamente per ogni stato non si riporta il valore di tutti gli attributi, ma solamente di quelli che lo caratterizzano.

Identificazione dei servizi richiesti Si possono distinguere due tipi di servizi.

- **algoritmicamente semplici**: normalmente non sono mostrati nel progetto del sistema in esame, ed agiscono sugli oggetti fornendo funzionalità standard. Esempi tipici sono il servizio per la creazione delle istanze ('create'), per il loro rilascio ('release'), per l'accesso al valore degli attributi dall'esterno ('access').
- **algoritmicamente complessi**: Sono specifici dell'applicazione in via di sviluppo e si possono dividere in due categorie:
  - *calcolo* ('*calculate*'): calcolano risultati a partire dai valori degli attributi dell'oggetto cui appartengono;
  - *controllo* ('*monitor*'): eseguono operazioni su sistemi esterni (altri oggetti) ed eventualmente reagiscono in qualche modo alle informazioni ottenute. Ci possono essere servizi in appoggio come quelli per inizializzare i sistemi sotto controllo.

Identificazione delle `Message Connection` Servono a modellizzare il fatto che un oggetto necessita di invocare un servizio di un altro per ottenere delle elaborazioni. La connessione può essere tra un oggetto ed una classe, quando il servizio da invocare sia a livello classe (ad esempio quello per la creazione di nuove istanze).

L'esame delle `Message Connection` consente all'analista di individuare le dipendenze di elaborazione tra le varie parti del sistema ottenendo un valido aiuto per effettuare eventuali divisioni in moduli.

Ricordando la natura incrementale e ricorsiva della metodologia ad oggetti, si deve tener presente che altre connessioni potranno essere individuate passando a realizzare i servizi.

Molto spesso il ricevente di un messaggio restituisce al mittente un valore come risultato delle elaborazioni compiute. Quando un oggetto manda un messaggio ad un altro, esso può rimanere bloccato fino a che l'esecuzione del metodo [2] invocato non è teminata (comunicazione *sincrona*). Oppure esso può continuare la propria esecuzione (comunicazione *asicrona*), però in questo caso deve essere previsto un protocollo per notificare al mittente che il servizio invocato è terminato, e passargli eventualmente il valore di ritorno.

Nel modello si può rappresentare l'interazione delle persone con il sistema mediante l'invocazione da parte di questi dei servizi. Cioè si può utilizzare un'icona per rappresentare le persone e tracciare delle 'Message Connection' tra questa e gli oggetti, anche se in realtà ci saranno delle interfacce che comunicano da una parte con le persone, e dall'altra con gli oggetti del sistema invocandone i metodi.

Per individuare più facilmente le 'Message Connection' ci si può chiedere per ogni oggetto:

- *da* quali oggetti necessita servizi;
- *a* quali oggetti è utile che fornisca servizi.

Si segue poi ogni 'Message Connection' collegata all'oggetto in esame e si ripetono le suddette considerazioni.

Di notevole utilità è anche analizzare i flussi dell'esecuzione ('control thread') seguendone i passaggi da un oggetto all'altro mediante le connessioni. In questo modo si può controllare che siano rispettati vincoli di tipo temporale, soprattutto nei sistemi in tempo reale ('real-time').

Specifica dei servizi I servizi possono essere specificati mediante due strumenti.

- Una *descrizione* a parole, più o meno formalizzata, può essere inclusa nella definizione della classe cui appartiene il servizio. Questa descrizione può eventualmente contenere uno pseudo-codice che meglio illustra il flusso delle azioni durante l'esecuzione. Inoltre possono essere evidenziati i dati in ingresso e quelli prodotti in uscita dal servizio, eventuali condizioni di inizio e terminazione, nonchè vincoli sulla modalità e la tempistica dell'esecuzione.

[2] *Metodo* è tipico della nomenclatura dell'orientamento ad oggetti, anche se non è usato nella terminologia Coad/Yourdon, ed è equivalente a servizio.

- Una '*Service Chart*' (una sorta di diagramma a blocchi) può essere associata ad ogni servizio. Questa permette di rappresentare il flusso delle operazioni e di definire comportamenti dipendenti dallo stato, utilizzando delle condizioni sull'esecuzione dei blocchi.

Se si hanno servizi che sono disponibili solamente in alcuni degli stati in cui si trova l'oggetto di cui fanno parte, può essere utile realizzare delle *tabelle servizi/stati* ('*Services/States table*') che riportano per ogni servizio gli stati in cui esso è utilizzabile.

6.5 Limiti del formalismo e convenzioni adottate

Nella progettazione di S^3 si è utilizzata la metodologia Coad/Yourdon per cui DECdesign fornisce supporto. Si è però notato che tale notazione non è del tutto soddisfacente per la modellizzazione di processi software. Uno dei suoi limiti principali è la mancanza di una rappresentazione del livello delle istanze, cioè di un'adeguata notazione per indicare i singoli oggetti e le varie connessioni e interazioni tra essi. All'opposto si può evidenziare l'assenza di una meta-rappresentazione del modello, cioè la possibilità di definire attributi e servizi di livello classe, nonchè di connessioni tra le classi invece che tra gli oggetti.

Mentre la rappresentazione esplicita del livello delle istanze non si è rivelata particolarmente importante nella modellizzazione dei processi software, di notevole rilievo si può considerare la possibilità di esprimere le suddette entità di livello classe. Va comunque tenuto presente che attributi, servizi e connessioni di livello classe non sono tanto importanti nella proggettazione, quanto nell'implementazione del modello per renderla più veloce ed efficiente.

Si è comunque riusciti a trascurare questi aspetti nella progettazione del modello di processo poiché le funzioni svolte da attributi e servizi di livello classe si possono considerare dettagli implementativi che non interessano questa prima fase. Per quanto riguarda le connessioni, si è supposto che le `Instance Connection` proprie della notazione Coad/Yourdon rappresentino sia un legame tra oggetti delle classi collegate, che un legame tra le classi stesse e da esse ispezionabile.

Va tenuto presente che l'implementazione sarebbe possibile anche senza

disporre di queste entità di livello classe, però sarebbe meno efficiente come verrà messo in evidenza in molte occasioni.

Inoltre, a livello di progetto, si è considerato che qualunque `Instance Connection` sia bidirezionale, cioè che ciascuno degli oggetti (e classi) collegate, conosca l'altro. Ciò non è sempre necessario, come viene riflesso dall'implementazione, ma sembra poco vantaggioso modificare la notazione in modo da esprimere questa differenza tra le connessioni a livello di modello. In modo del tutto analogo non è detto che per qualsiasi connessione sia indispensabile una corrispondenza sia tra gli oggetti che tra le classi, ma non sembra particolarmente vantaggioso differenziare i vari casi complicando il modello o quanto meno la notazione utilizzata.

Nella parte III sarà presentato ed illustrato il progetto di S^3 e nel commentare le varie connessioni sarà specificato il loro utilizzo, e quindi se debbano mettere il relazione istanze, classi o entrambe. Nella spiegazione dell'implementazione del modello saranno fatti cenni anche alla direzionalità delle connessioni. Nel progetto di S^3 le `Instance Connection` sono state tracciate utilizzando un attributo in ognuna delle due classi interessate a cui collegare la connessione come spiegato nella sezione A.3. Ciò permette di evidenziare, mediante il nome di questi attributi, il compito della connesione ed il ruolo ricoperto da ciascuno dei due partecipanti. In questa trattazione le connessioni saranno identificate mediante il nome dei due attributi che collegano separati da una barra (ad esempio `attributo1/attributo2`)

7

Metodologia Booch di progettazione ad oggetti

Si è deciso di dare alcuni accenni riguardo alla metodologia di progettazione proposta da Booch perché essa è molto completa, ma soprattutto perché può essere un valido termine di confronto nella valutazione della metodologia Coad/Yourdon. Dopo aver elencato i principi che la caratetrizzano, olter a quelli tipici dell'orientamento agli oggetti, si sono brevemente illustrate le fasi che costituiscono la progettazione.

7.1 Principi fondamentali

La metodologia di progettazione ad oggetti proposta da Booch si basa su alcuni concetti la cui presenza è considerata indispensabile ('*major entity*') [Boo91]:

- astrazione
- incapsulamento ('encapsulation')
- modularità
- gerarchia

Ci sono poi altri concetti importanti, ma non indispensabili ('*minor entity*'):

- tipizzazione ('typing')

- concorrenza
- persistenza

Tutti questi principi sono stati descritti nella sezione 5.3.

7.2 Fasi della metodologia

Alla base della progettazione ad oggetti ('*object oriented design*') sta la capacità di identificare nel dominio in esame le classi, e quindi gli oggetti, e le relazioni esistenti tra queste, più opportune per modellizzare il problema.

La *classificazione* è un principio di qualunque scienza e ci sono vari metodi più o meno affermati mediante cui applicarla. In generale comunque non si è in grado di stabilire quale sia la soluzione migliore e neppure se questa esista.

In ogni caso la classificazione ha una natura intrinsecamente *incrementale ed iterativa* (procedere per passi successivi, ritornando eventualmente sulle fasi già considerate, con raffinamenti graduali) da cui è caratterizzato l'approccio ad oggetti.

In particolare può succedere che durante la progettazione si definiscono delle classi con una certa struttura e legate da certe relazioni che, in fase di implementazione (programmazione), si rivela necessario modificare. Infatti quando si costruiscono i clienti di una certa classe possono risultare evidenti cambiamenti molto importanti alla classe in questione, oppure si può decidere di creare nuove classi o condensarne alcune in una sola, oppure ancora modificare le relazioni tra classi esistenti.

Soprattutto nello sviluppo di grossi sistemi, si apprezza la validità di un approccio di tipo incrementale ed iterativo che è tipico della progettazione ad oggetti. Altro apetto significativo è che nei raffinamenti successivi si può anche operare a livelli di astrazione differenti. Infatti [Cur89] afferma che "un buon progettista ad oggetti è in grado di lavorare contemporaneamente a diversi livelli di astrazione e di dettaglio".

Passi concettuali Secondo Booch l'approccio incrementale ed iterativo tipico della progettazione ad oggetti si basa sui passi concettuali di seguito elencati.

- Si definiscono delle astrazioni (classi) con una certa struttura ed alcune elementari relazioni tra esse.

- Si stabiliscono meccanismi che usano queste astrazioni e ne permettono la cooperazione.

- Progettando i dettagli dei meccanismi ci si rende conto delle modifiche da fare sulle classi e sulle relazioni tra queste. Infatti a questo livello ci si rende conto della necessità di nuove relazioni tra le classi o di caratteristiche comuni ad un certo numero di esse.

- Si fanno le modifiche necessarie e si torna ai passi precedenti fino a che non si ottiene un insieme di diagrammi di classi e di oggetti che danno probabilmente viste differenti del sistema, ma che sono coerenti e consistenti perché sono evoluti incrementalmente attraverso modelli stabili e funzionanti anche se sempre più complessi.

Strumenti Come accennato in precedenza gli strumenti, e nello stesso tempo i prodotti, della progettazione ad oggetti, sono i *diagrammi delle classi* e quelli *degli oggetti*. I primi contengono le varie classi e le relazioni esistenti tra essi [Boo91]. Gli altri gli oggetti ed i messaggi che questi si scambiano vicendevolmente.

Delle classi ed oggetti costituenti i diagrammi si possono specifiacare caratteristiche e comportamento mediante opportune descrizioni ('*templates*'). Va tenuto presente che, come ci sono differenti orientamenti nella progettazione ad oggetti, ci sono anche notazioni differenti per rappresentare i prodotti della progettazione, uno dei quali è appunto quello descritto in [Boo91].

Altri tipi di diagrammi sono quelli dei moduli e dei processi che fanno però riferimento alla struttura fisica del prodotto in via di sviluppo.

Procedura operativa Ai passi concettuali descritti in precedenza, tenuto conto anche dei suddetti strumenti e prodotti, corrispondono dei passi operativi da effettuare quando si stia seguendo una progettazione ad oggetti [Boo91, pag. 90].

- Identificare classi ed oggetti ad un certo livello di astrazione (quello attualmente considerato).

- Identificare la semantica di alcune di queste classi ed oggetti.

- Identificare le relazioni tra queste classi ed oggetti.

- Implementare le classi e gli oggetti nel caso di una effettiva produzione di software, oppure semplicemente specificarne in modo dettagliato gli attributi ed le procedure.

Questo procedimento è però iterativo per cui, come detto sopra, arrivati all'implementazione, ci si può rendere conto di altre classi ed oggetti da aggiungere o di modifiche da fare a quelli già esistenti.

Analisi e progettazione Molto difficile è stabilire il confine tra analisi e progettazione che è molto labile, qualunque sia l'approccio utilizzato. Questo vale ancor più con un approccio ad oggetti. In ogni caso secondo Booch si può dire che l'analisi ad oggetti consti nell'individuare un insieme di classi ed oggetti derivati direttamente dal vocabolario tipico del dominio in questione.

La fase di progettazione si occupa invece di modellizzare in modo piuttosto completo tutte le entità che hanno una qualche rilevanza nelle responsabilità del sistema in esame. Non è facile stabilire quando questa fase di progettazione sia da dichiararsi conclusa, pur dopo aver fatto un certo numero di iterazioni che prevedono anche il passaggio alle altre fasi–siano esse precedenti o successive.

Un valido mezzo di decisione può essere il fatto che l'implementazione delle entità rilevanti possa essere fatta componendo tra loro le astrazioni già individuate senza bisogno di considerarne altre. Cioè ci si può fermare quando le astrazioni fatte sono sufficientemente semplici da non dover richiedere ulteriori suddivisioni e l'uso di altri livelli di astrazione.

Vediamo ora nel dettaglio i passi del procedimento di progettazione ad oggetti secondo [Boo91] che, come più volte affermato, non sono strettamente sequenziali, ma quando già si lavora sui passi successivi si può tornare su quelli precedenti per migliorarne il prodotto secondo quello che è l'approccio incrementale ed iterativo tipico del modello ad oggetti.

7.2.1 Identificazione delle classi

Questo passo iniziale, almeno la prima volta che lo si affronta, è strettamente connesso all'analisi. Avendo raggiunto una buona conoscenza del dominio del problema grazie all'analisi fatta su di esso, si può passare ad individuare quelle che sono le *classi candidate* ed i meccanismi mediante cui queste interagiscono per ottenere determinati comportamenti.

Quindi questo passo non è ben distinto dall'analisi che ha come scopo la definizione dei principali oggetti che intervengono nel dominio del problema e delle relazioni logiche ed interazioni tra questi. Tutto ciò si ottiene mediante una conoscenza di tale dominio, che non è però un'attività indipendente, ma viene approfondita anche quando si cominciano ad individuare gli oggetti. Questi oggetti sono raffinati in questo primo passo della progettazione.

Le classi individuate in questa fase della progettazione sono state dette 'candidate' perché nei raffinamenti successivi alcune potranno essere eliminate ed altre nuove aggiunte, alcune raggruppate tra loro ed altre suddivise in un maggior numero di classi.

Questo primo passo della progettazione può produrre una semplice lista di classi ed oggetti, o anche un diagramma delle classi con alcune relazioni, ancora di tipo generale, tra esse. Inoltre per alcune entità può essere utile cominciare a scrivere delle informazioni più o meno dettagliate ('*template*') in forma descrittiva.

7.2.2 Identificazione della semantica delle classi

In questa fase si devono prendere in considerazione le classi precedentemente definite per determinarne più nel dettaglio la semantica ed il ruolo. Durante questo passo si vedono le classi dal punto di vista dell'interfaccia e delle interazioni con le altre.

A questo punto il processo diviene iterativo perché specificare le interazioni tra due oggetti, può rendere evidenti alcune modifiche necessarie alle interfacce di questi o addirittura alla loro struttura.

Inoltre si rifiniscono i 'templates' dei vari elementi in gioco che erano stati abbozzati al passo precedente continuando a specificare sempre più nel dettaglio la semantica delle varie classi e degli oggetti individuati.

I prodotti di questa seconda fase del progetto sono altri diagrammi di classi o di oggetti per documentare i nuovi meccanismi introdotti o per specificare più nel dettaglio quelli già esistenti.

7.2.3 Specifica delle interazioni

Si tratta in pratica di una estensione del passo precedente in cui si specifica di che tipo sono le relazioni tra le classi e le interazioni tra gli oggetti che si erano identificate prima, seppur in modo piuttosto generico.

Nel dettagliare i meccanismi che utilizzano o sono utilizzati dagli oggetti, se ne deve specificare in modo il più possibile completo la semantica statica e dinamica.

È in questa fase che ci si deve maggiormente rendere conto delle analogie tra le classi che permettono magari di semplificare la struttura del sistema riducendone il numero di elementi. Dunque si rifiniscono i diagrammi delle classi e degli oggetti con i nuovi raffinamenti o le analogie riscontrate, realizzando eventuali raggruppamenti o variazioni alle relazioni.

Analogamente si deve cercare di generalizzare il più possibile i meccanismi, in modo da sfruttarli in contesti simili facendoli condividere. Questo porta come conseguenza una serie di cambiamenti ai diagrammi degli oggetti per introdurvi le modifiche agli scambi di messaggi tra oggetti che implementano i comportamenti voluti. Inoltre, dopo aver definito più nel dettaglio il modo in cui cooperano gli oggetti di cui si sia già individuata l'interazione, si devono considerare le varie coppie di oggetti per individuare eventuali nuove collaborazioni specificandone le modalità.

Sempre a questo livello vanno fatte le scelte di visibilità qualora si scelga un approccio che faccia uso della modularità.

7.2.4 Definizione del corpo di classi ed oggetti

Fino a questo momento si è curata solamente una visione esterna di classi ed oggetti, sia questa la loro interfaccia o il loro comportamento. È comunque arrivato il momento di considerare anche la rappresentazione interna di queste entità specificandone gli attributi che ne determinano lo stato ed i metodi che operano si di essi (anche quelli non visibili dall'esterno).

È proprio durante questo passo che si esplicita maggiormente la metodologia incrementale ed iterativa. Infatti, proprio mentre si curano questi aspetti, ci si può rendere conto di alcune modifiche necessarie negli ambiti curati nelle prime fasi della progettazione e quindi si ritorna su di essi per modificarli o anche semplicemente rifinirli.

È molto importante questo concetto perché non è indispensabile approfondire le varie fasi in modo assolutamente completo con elevato dispendio di tempo ed energie mentali, ma ci si può tornare sopra scendendo ogni volta ad un livello di dettaglio sempre più elevato fino ad ottenere, dopo un certo numero di iterazioni attraverso questi quattro passi, un progetto completo e dettagliato.

In questa quarta fase si modificano e rifiniscono pesantemente i 'templates' dei vari elementi che entrano a far parte del progetto.

Parte III

Progetto dell'ambiente S^3

8

Aspetti generali di S^3

In questo capitolo, dopo aver introdotto le principali caratteristiche di S^3, viene illustrata la struttura generale del progetto dell'ambiente di simulazione. Quindi sono descritti i principali meccanismi da cui è caratterizzato S^3, quali l'istanziazione delle sottoattività, la gestione dell'esecuzione e coordinazione delle attività, la gestione delle varie entità coinvolte nel processo e dei loro rapporti con le attività. Sono forniti anche alcuni accenni sulla condivisione ed il versionamento dei dati.

8.1 Introduzione

Uno degli scopi di questo lavoro è stato la realizzazione di S^3, un ambiente per la simulazione dei modelli di processo. S^3 è stato progettato con la metodologia ad oggetti Coad/Yourdon avvalendosi di DECdesign. Questo è uno strumento automatico che fornisce supporto per la progettazione secondo alcune differenti metodologie, tra cui anche quella Coad/Yourdon. Per l'implementazione di S^3 ci si è serviti di Objectworks(r)/Smalltalk, un linguaggio ad oggetti che è largamente utilizzato nella prototipizzazione.

Un tipico ambiente PM offre:

- un linguaggio per la modellizzazione dei processi (PML),
- degli schemi predefiniti
- degli strumenti di supporto (*PM tools*).

In S^3:

- Smalltalk-80 costituisce il PML;
- le classi che possono essere utilizzate nella creazione di modelli di processo costituiscono gli schemi predefiniti messi a disposizione per la modellizzazione;
- l'ambiente di sviluppo offerto da Objectworks(r)/Smalltalk e le interfacce per l'istanziazione del modello di processo, costituiscono gli strumenti di supporto all'implementazione e istanziazione dei processi.

La realizzazione di S^3 non porta solamente alla creazione di un ambiente per la simulazione dei processi software, ma anche a proporre una tecnica orientata agli oggetti per la modellizzazione dei processi.

8.2 Struttura del nucleo di S^3

S^3 è costituito da un insieme di classi che, da un lato sono lo schema predefinito dell'ambiente PM, dall'altro realizzano il motore per la simulazione dei processi. Infatti queste classi vengono utilizzate, direttamente o come generalizzazione da cui derivare per ereditarietà classi più specifiche, per la modellizzazione dei processi. Allora il modello di processo che si ottiene, include le funzionalità offerte da queste classi ed è grazie a queste funzionalità che è possibile farne una simulazione. Tutto ciò è perfettamente in linea con i principi dell'orientamento agli oggetti secondo cui gli oggetti inglobano le funzionalità che gli occorrono per autogestirsi senza che siano loro garantite dal sistema che li ospita.

Il progetto del nucleo ('*kernel*') di S^3 è costituito da quattro soggetti, individuati a partire da altrettante classi, che sono le più generali tra tutte quelle contenute nel rispettivo soggetto.

1. `Task` contiene le classi create per modellizzare le attività costituenti il processo software.
2. `Role` contiene le classi per modellizzare i ruoli ricoperti dalle persone che intervengono nel processo software.
3. `Data` contiene il modello dei dati; le istanze delle sue classi vengono utilizzate per rappresentare i prodotti delle varie attività e per definire la configurazione del prodotto del processo.

4. `Tool` contiene le classi per la rappresentazione degli strumenti automatici utilizzati per portare a termine le varie attività coinvolte nel processo.

Del nucleo di S^3 fa parte anche la classe `Person`; essa non è inclusa in nessun soggetto e serve per modellizzare le persone che intervengono nel processo software.

Le classi generali descritte in precedenza sono collegate dalle connessioni mostrate nella figura 8.1; per realizzare il modello di uno specifico processo queste connessioni dovranno essere dettagliate tracciandole tra le sottoclassi corrispondenti. Nella figura 8.1 non è mostrato lo strato dei servizi per non complicare troppo il disegno e perché i servizi non rivestono particolare interesse a questo punto della trattazione. Nel seguito saranno illustrati nel dettaglio gli attributi, le connessioni ed i metodi di ognuna di queste classi e di alcune loro specializzazioni.

Le classi che costituiscono il nucleo di S^3 forniscono le funzionalità per realizzare i meccanismi su cui si basa la simulazione processi software; avere a disposizione tali meccanismi è indispensabile per la simulazione dei modelli di processo. I modelli di processo sono realizzati creando specializzazioni di queste classi e quindi estendo ad esse, per ereditarietà, le funzionalità che sono proprie di S^3.

Un ambiente di modellizzazione dei processi software deve mettere a disposizione una libreria di classi predefinite che possono essere utilizzate per la realizzazione dei modelli di processo, sia creandone specializzazioni, sia includendole direttamente. Deve quindi essere possibile riutilizzare, se necessario, qualunque classe sia nota al sistema, cioè anche quelle che non sono di utilità generale, ma che sono state create per la modellizzazione di qualche specifico processo. Questo riutilizzo non interessa solo la progettazione del modello, ma anche l'implementazione, ed è ancora più importante nel momento in cui si usi un approccio ad oggetti che fa del riutilizzo uno dei suoi principali punti di forza. Allora un ambiente completo ed efficiente per la modellizzazione dei processi software dovrà comprendere uno strumento automatico di sussidio alla modellizzazione che permetta di accedere alle librerie di classi predefinite per includerle nei modelli, e dia la possibilità di creare nuove classi.

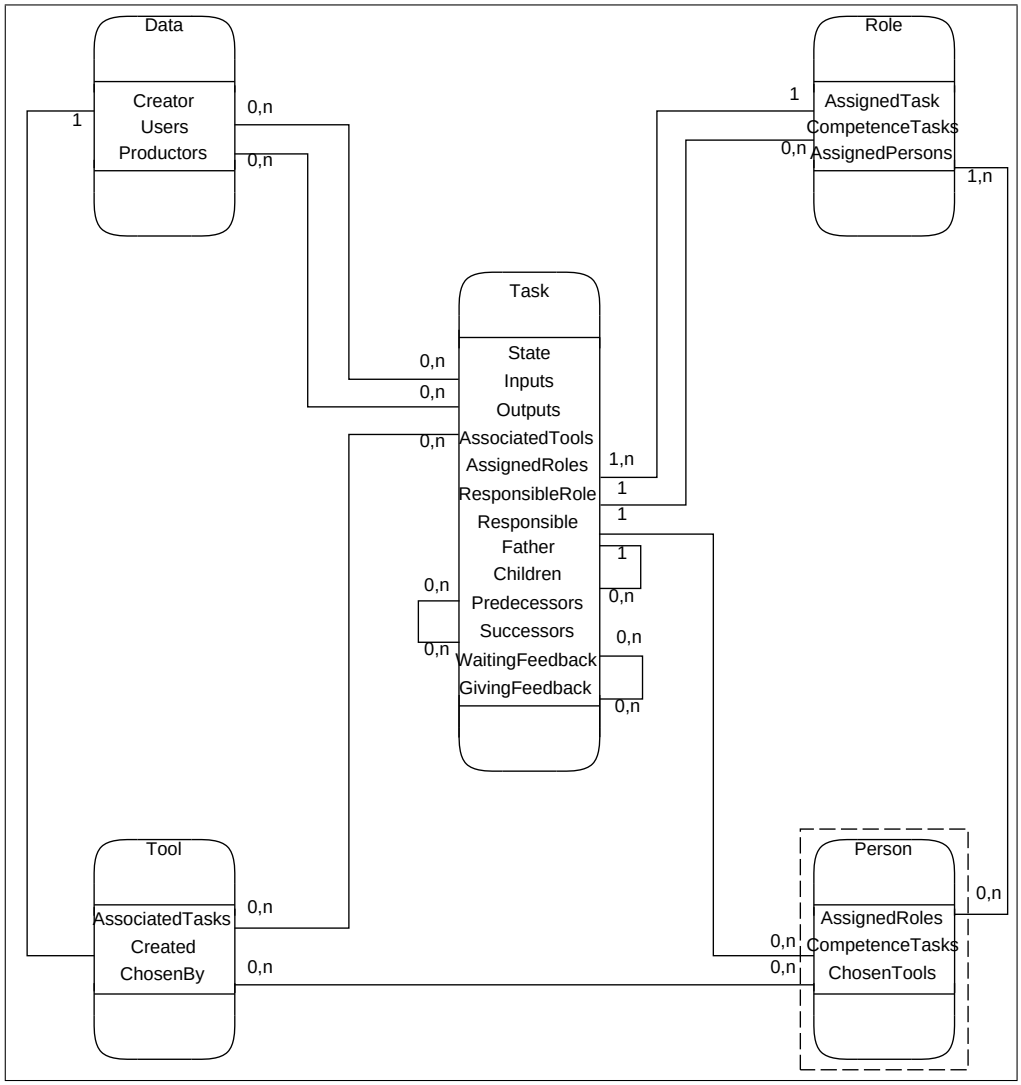

Figura 8.1. Classi generali

8.3 Istanziazione delle sottoattività

Le attività che sostituiscono un modello di proceso software si dividono in *atomiche* e *composte*. Quelle atomiche consistono nell'esecuzione di strumenti automatici o nell'interazione con persone. Le attività composte delegano alcuni dei loro compiti ad attività di più basso livello; ciò è espresso dicendo che esse si decompongono in sottoattività. Nella realtà tale decomposizione

avviene in maniera incrementale al sorgere di condizioni quali la disponibilità di risorse umane e non, la terminazione di altre attività, ecc.

S^3 realizza l'*istanziazione incrementale*, cioè un'attività composta, non crea le sue sottoattività fino a che queste non siano pronte ad essere eseguite. Nella maggior parte degli ambienti PM ciò non accade.

Poiché tra le attività generate da uno stesso padre si possono avere vincoli di precedenza, non è detto che queste possano essere eseguite tutte appena istanziate. Il meccanismo di istanziazione delle sottoattività potrebbe creare tutti gli oggetti delle sottoclassi di `Task` che le rappresentano e questi resterebbero inattivi fino a che non si verafichino le opportune condizioni per l'esecuzione delle attività corrispondenti.

Una tale politica però, porterebbe ad un'inutile occupazione di risorse per tempi anche molto lunghi; infatti le attività tipiche dei processi software possono avere durate notevoli, anche di anni, per cui non è tollerabile creare un'oggetto che non ha alcuna utilità per un tempo così lungo.

L'attuazione (simulazione in S^3) del modello di processo è controllata facendo in modo che ogni attività sia in grado di decidere quando può cambiare stato. Poiché lo stato di un'attività dipende anche da quello delle altre, durante l'attuazione si ha uno scambio di messaggi che ha lo scopo di mantenere ciascun oggetti aggiornato sullo stato degli altri. Dunque qualsiasi attività istanziata, anche se non in esecuzione, deve pertecipare a questo scambio di messaggi, ma tutti i messaggi inviati ad attività che non possono ancora cominciare la loro esecuzione sono perfettamente inutili rappresentando un carico superfluo per il sistema.

La scelta dell'utilizzo dell'istanziazione incrementale ha anche una conseguenza sul meccanismo di assegnazione delle persone: l'assegnazione coincide temporalmente con l'istanziazione delle attività. Infatti, poiché l'istanziazione avviene solo quando l'attività può essere eseguita, la persona può essere assegnata all'attività appena quest'ultima è creata. Questa soluzione non potrebbe essere adottata se tutti gli oggetti che rappresentano le sottoattività fossero istanziati subito; infatti, non sapendo dopo quanto tempo comincia l'esecuzione, non ha senso assegnare delle persone, perché non si conosce assolutamente il momento in cui saranno chiamate ad operare. Questo fatto avrebbe come conseguenza l'impossibilità di tenere in qualche modo in considerazione il carico di lavoro delle persone coinvolte nel processo.

Il vantaggio dell'istanziazione immediata di tutti i figli deriva dal fatto che il padre è liberato dall'onere di stabilire quando questi siano pronti per

essere eseguiti e possano quindi essere istanziati. Nella sezione 10.2 viene spiegato più nel dettaglio il meccanismo per l'istanziazione incrementale delle sottoattività e viene messo in evidenza il fatto che il sovraccarico del padre è del tutto accettabile.

8.4 Esecuzione delle attività

Dopo la loro creazione, le attività permangono in stati nei quali reagiscono ad eventi ed intraprendono azioni; uno di questi stati caratterizza l'esecuzione.

Nella realtà le attività che costituiscono un processo per la produzione del software sono concorrenti, pur esistendo vincoli che impongono un ordine tra le esecuzioni di alcune attività. In S^3 le attività sono modellizzate da oggetti attivi, però questo fatto non è espresso esplicitamente nel progetto perché la metodologia Coad/Yourdon non fornisce una particolare rappresentazione per gli oggetti attivi.

Nel seguito sono elencate le azioni che tipicamente costituiscono l'esecuzione di un'attività.

- Attesa della disponiblità dell'utente ad interagire.
- Esecuzione di interfacce per l'interazione con l'utente.
- Esecuzione di strumenti automatici.
- Istanziazione di sottoattività e gestione della loro terminazione.

8.5 Coordinazione delle attività

L'esecuzione delle attività è vincolata da precondizioni che stabiliscono un ordine di precedenza. In S^3 queste precondizioni sono imposte mediante connessioni tra gli oggetti del modello e sono controllate tramite scambio di messaggi.

Le condizioni riguardano entità coinvolte nel processo (ad esempio altre attività, oppure dati) e quindi rappresentate nel modello; la connessione permette di individuare quali di queste entità vincolano l'esecuzione di un'attività. Alla connessione è assegnata una semantica che esprime lo stato in cui deve trovarsi l'entità collegata affinchè la condizione associata si possa considerare verificata.

Nel progetto di S^3 le connessioni sono tracciate sempre tra due attributi degli oggetti che collegano (vedi appendice A.3). Tali attributi individuano univocamente la connessione specificando anche i ruoli ricoperti dai due oggetti che partecipano alla connessione. La semantica della connessione è individuata dal nome della connessione stessa, cioè dagli attributi a cui sono attaccati gli estremi.

8.6 Gestione dei dati

I dati prodotti ed utilizzati dalle attività rivestono due ruoli fondamentali nella modellizzazione dei processi software:

1. permettono di scambiare informazioni tra le attività;
2. sono parte del prodotto finale del processo; esso è un insieme di dati prodotti da alcune delle attività del processo software.

S^3 prevede una rappresentazione dei dati che permette, da un lato di esprimere a livello di modello di processo, quali sono le attività che producono e utilizzano ciascun dato, dall'altro di gestire la produzione e l'utilizzo dei dati, all'atto della simulazione del processo.

La gestione dei dati prodotti ed utilizzati dalle attività è di tipo gerarchico, cioè legata alla gerarchia di decomposizione delle attività. Ciò implica che i dati manipolati da un'attività devono coincidere, o essere in qualche modo correlati, con quelli utilizzati dall'attivit'a composta che ha istanziato l'attività in questione.

Inoltre la rappresentazione dei dati prevista da S^3 consente di definire la struttura del prodotto del processo, sia in modo generale a livello di modello, sia in modo più specifico al momento della simulazione del processo.

8.7 Gestione degli strumenti

Un processo per la produzione del software richiede l'esecuzione di strumenti automatici che aiutano le persone nello svolgimento dei loro compiti. L'ambiente PM si deve occupare di fornire gli strumenti utili per l'esecuzione delle attività, ma soprattutto li deve eseguire sulla macchina della persona interessata e deve fare in modo che i vari strumenti lavorino sui dati opportuni. Queste funzionalità sono legate a problematiche di esecuzione remota

ed incapsulamento degli strumenti e devono essere del tutto trasparenti agli utenti.

Le responsabilità direttamente legate alle problematiche di esecuzione distribuita ed incapsulamento degli strumenti, possono essere delegate ad un substrato su cui l'ambiente PM è realizzato, in modo che i dettagli di questi meccanismi non interessino l'implementazione dell'ambiente in questione. Con un approccio orientato agli oggetti questa suddivisione dei compiti e questa non visibilità dei dettagli risultano ancora più immediate perché i meccanismi di esecuzione degli strumenti sono responsabilità degli oggetti che li rappresentano.

S^3 si occupa di simulazione dei processi e quindi non è necessario che esegua realmente gli strumenti; però deve tenere in conto le problematiche tipiche della gestione degli strumenti che sono legate ai dati che essi devono manipolare ed alle attività che ne fanno uso.

L'ambiente presentato in questa trattazione fornisce una struttura che permetta di esprimere a livello di modello quali tipi di strumento possano operare su di un certo tipo di dato; inoltre S^3 prevede meccanismi che permettano di stabilire quale strumento utilizzare tra quelli di un certo tipo.

Nel progettare S^3 si è supposto che un dato, una volta creato da un certo strumento automatico, possa essere manipolato solo da quello stesso strumento, e non eventualmente da altri dello stesso genere. Questa imposizione è indispensabile in alcuni casi (quando il formato del dato è strettamente dipendente dal programma che lo ha generato), ma in altri può rappresentare una notevole restrizione (ad esempio un testo prodotto ad un semplice 'editor' può essere modificato da un qualsiasi altro 'editor'). Con una differente gestione dei dati e degli strumenti automatici è possibile evitare questa restrizione, ma questo non è uno degli obiettivi primari di questo lavoro.

S^3 mette a disposizione meccanismi per specificare quali strumenti possano essere utilizzati da un'attività per realizza la propria esecuzione. La gestione di questi aspetti è di tipo gerarchico, cioè quando un'attività si decompone sceglie gli strumenti da mettere a disposizione delle sottoattività tra quelli che le erano stati associati.

8.8 Gestione delle persone e dei loro ruoli

S^3 prevede che ad ogni attività sia associata almeno una persona che è responsabile di tale attività. Questo significa che alla persona in questione

può essere richiesto di intervenire operando scelte che consentono di portare a termine l'esecuzione dell'attività, o utilizzando strumenti automatici per manipolare i dati che l'attività deve produrre.

L'interazione tra il sistema e le persone avviene mediante interfacce il cui funzionamento sarà illustrato quando tratteremo l'implementazionedi S^3. Infatti, come accennato in precedenza, il progetto prende in considerazione solamente il componente del dominio del problema (sezione 6.2).

Le attività modellano l'interazione con la persona mediante uno scambio di messaggi tra gli oggetti che rappresentano le attività e quelli che rappresentano le persone. Nella realizzazione di S^3 tali messaggi sono tradotti nell'invocazione di metodi delle interfacce, le quali causano degli eventi visibili all'utente; a tali eventi l'utente può rispondere tramite gli strumenti messi a disposizione dall'interfaccia stessa.

Ad ogni attività composta, oltre al responsabile, è associato un insieme di persone che sono a disposizione per poter essere assegnate alle sottoattività. L'associazione di queste persone all'attività avviene nel momento in cui quest'ultima è istanziata.

S^3 prevede un meccanismo di associazione delle persone alle attività basato sulla gerarchia di decomposizione delle attività e sui ruoli ricoperti dalle persone.

8.8.1 Assegnazione gerarchica

Il meccanismo è gerarchico perché le persone che possono essere assegnate ad un'attività, sono solamente quelle a disposizione dell'attività composta che la ha generata. Quando una persona è messa a disposizione di un'attività, vengono specificati i ruoli che essa ricopre; se una persona è assegnata a più attività, non è detto che per tutte queste ricopra gli stessi ruoli. Inoltre una persona messa a disposizione di una sottoattività, non è detto che ricopra per essa tutti i ruoli che ricopre per l'attività padre, ma può essere sufficiente un sottoinsieme di tali ruoli.

L'assegnazione delle persone alle sottoattività è fatta al momento dell'istanziazione di queste ultime, ed è affidata all'attività composta che le genera. Talvolta l'assegnazione richiede l'intervento di una persona che deve scegliere in un elenco di candidati; quando ciò si verifica, il compito è affidato al responsabile dell'attività composta che ha istanziato la nuova sottoattività per cui sono necessarie le assegnazioni.

8.8.2 Assegnazione basata sui ruoli

I ruoli ricoperti dalle persone rivestono importanza fondamentale in due momenti dell'assegnazione delle persone ad un'attività.

- Il responsabile di un'attività deve ricoprire un ruolo ben preciso che è espresso a livello del modello di processo secondo le modalità che saranno descritte nella sezione 9.2.1. Quando un'attività composta si decompone, ad ogni sottoattività deve essere assegnato il rispettivo responsabile. Questo va scelto tra le persone assegnate, nel ruolo imposto per il responsabile, all'attività composta.

- Un'attività composta può assegnare alle sue sottoattività solo le persone che ricoprono uno dei ruoli imposti dal modello del processo per il responsabile di tali sottoattività. Queste persone devono essere scelte tra quelle a disposizione dell'attività che si sta decomponendo rispettando così l'aspetto gerarchico del meccanismo.

8.8.3 Assegnazione non gerarchica

L'assegnazione gerarchica non è l'unica soluzione possibile. Si può considerare un'alternativa non gerarchica che vede un insieme di persone, ognuna delle quali ricopre uno o più ruoli, che possono svolgere la funzione di responsabile di qualunque attività che richieda uno dei ruoli ricoperti. Quindi una persona, ad esempio il 'project manager', è incaricata di associare le persone alle varie attività a mano a mano che queste ultime ne hanno bisogno.

Questa soluzione ha due svantaggi per chi è incaricato dell'assegnazione:

1. ha un notevole carico di lavoro che si può protrarre per tutta la durata del processo;

2. deve conoscere le attitudini di un gran numero di persone e le caratteristiche di molte attività, per poter eseguire le assegnazioni nel modo più oculato possibile.

8.9 La condivisione ed il versionamento dei dati

Le attività possono essere eseguite più volte e così nascono problemi su come gestire i dati che esse producono, perché un'attività può trovarsi a modificare un dato che un'altra sta utilizzando. Queste problematiche sono legate alla gestione delle configurazioni ('configuration management') e delle versioni ('version management'). In realtà l'integrità della configurazione del prodotto deve essere garantita dal processo stesso, ammesso che questo sia stato ben strutturato. Quindi può essere utile avere la possibilità di effettuare delle verifiche sul modello per controllare che esso mantenga la coerenza della configurazione del prodotto finale.

Indispensabile è la gestione delle versioni integrata con un'accurata gestione della condivisione dei dati da parte delle varie attività. Infatti è molto comune la situazione in cui il dato prodotto da una certa attività debba essere modificato da un'altra, magari nello stesso istante in cui una terza lo sta utilizzando.

La soluzione che sembra essere più semplice, e quindi più pratica e robusta, è quella di non modificare mai un dato una volta prodotto, ma di crearne sempre una nuova versione. Inoltre le attività che utilizzzano un dato, ne utilizzano in realtà una ben determinata versione, che non può essere modificata da altre attività. Quando la versione utilizzata da un'attività non è la più recente, deve essere il processo che, essendo stato definito bene, fa in modo che l'attività sia rieseguita sulla versione più recente. Se il processo è stato strutturato correttamente, alla fine della sua esecuzione ogni attività avrà agito sulla versione più recente dei dati in ingresso, producendo una versione aggiornata dei dati in uscita.

Nella progettazione di S^3 la gestione della condivisione e del versionamento dei dati non è stata affrontata esplicitamente perché si è supposta l'esistenza di un substrato che si fa carico di questi oneri. Inoltre l'utilizzo dell'orientamento agli oggetti ha permesso di delegare questi aspetti all'implementazione degli oggetti che rappresentano i dati come sarà descritto nella sezione 9.3.2.

9

Le classi del progetto di S^3

In questo capitolo sono riportate le varie classi che costituiscono il progetto ad oggetti di S^3. Non è stato possibile introdurre un diagramma delle classi globale perchè non si è riusciti a trovare un formato per adatto ad inserirlo nella trattazione in modo da essere comprensibile. Allora si è descritto ogni soggetto separatamente senza mostrare tutte le connessioni tra calssi che sono contenute in soggetti differenti.

9.1 Il soggetto `Task`

Ogni attività che entra a far parte del processo da modellizzare, è rappresentata da un'istanza di una sottoclasse di `Task`. in S^3 la classe `Task` raccoglie in sé tutte le caratteristiche delle attività, fornendo la definizione di tutti gli attributi e dei servizi necessari per l'esecuzione e la coordinazione delle attività.

Qualsiasi attività viene modellizzata come sottoclasse di `Task`, ma questa non è l'unica soluzione possibile. Se infatti si vuole dare un modello sofisticato delle attività senza complicare eccessivamente la singola classe `Task`, si possono definire sottoclassi astratte di `Task` ciascuna delle quali individua le caratteristiche tipiche di una certa categoria di attività (ad esempio quelle mostrate in figura 9.1). Per rappresentare le attività che effetivamente intervengono nel processo si utilizzano quindi classi che, mediante ereditarietà multipla, derivano dalle generalizzazioni che hanno caratteristiche tipiche dell'attività da modellizzare.

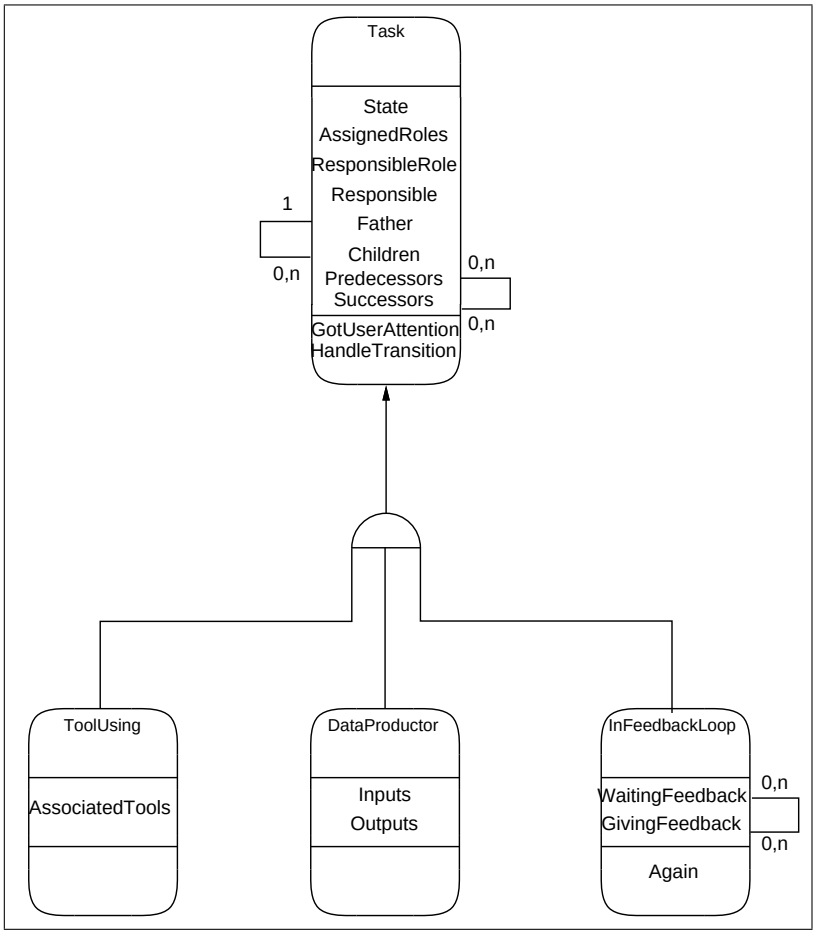

Figura 9.1. Sottoclassi generali di `Task`

In S^3 la classe `Task` (figura 9.2) raccoglie in sé caratteristiche e comportamenti che permettono di modellizzare qualsiasi attività, cioè attività in grado sia di decomporsi in sottoattività, sia di interagire con l'utente, sia di produrre i propri dati di uscita utilizzando strumenti automatici. Le attività più semplici, pur ereditando tutte queste funzionalità, non utilizzeranno alcuni dei servizi o degli attributi messi a disposizione; inoltre quando necessario, nelle sottoclassi potranno essere aggiunti o ridefiniti i metodi e gli attributi della classe `Task`.

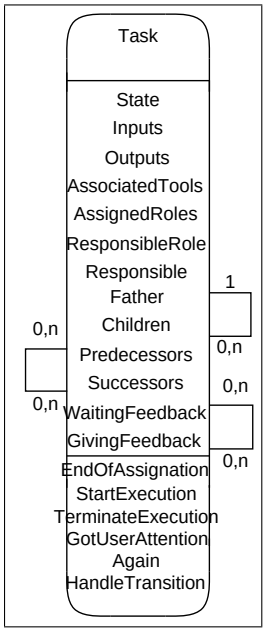

Figura 9.2. Classe `Task`

9.1.1 La gerarchia delle attività

Le attività composte si possono decomporre in sottoattività che sono dette *figlie*; in S^3 questa caratteristica è propria di qualsiasi istanza di una sottoclasse di `Task`, per cui non si ha una vera e propria distinzione tra attività composte ed atomiche. L'identificazione dei figli di un `Task` avviene grazie alla connessione che collega gli attributi `Father` e `Children` di due sottoclassi di `Task`. Questa 'instance connection' è definita per la classe `Task`, ma deve essere ridefinita tra le sue sottoclassi per specificare nel modello di che tipo sono le figlie di una certa attività.

Nella figura 9.3 è riportato un esempio di decomposizione gerarchica di un'attività che realizza il ciclo di scrittura, compilazione e costruzione dell'eseguibile ('edit-compile-link'). Tutte le classi riportate nella figura sono specializzazioni di `Task`, ma la struttura `Gen/Spec` non è stata tracciata per non complicare troppo il disegno. Oltre all'attività composta ed alle tre attività che costituiscono il ciclo, compare l'attività `AssignTasks` come figlia di `CicloECL`. `AssignTasks` è una sottoclasse predefinita di `Task`, ciò fa parte della struttura di classi di utilità generale messa a disposizione da S^3.

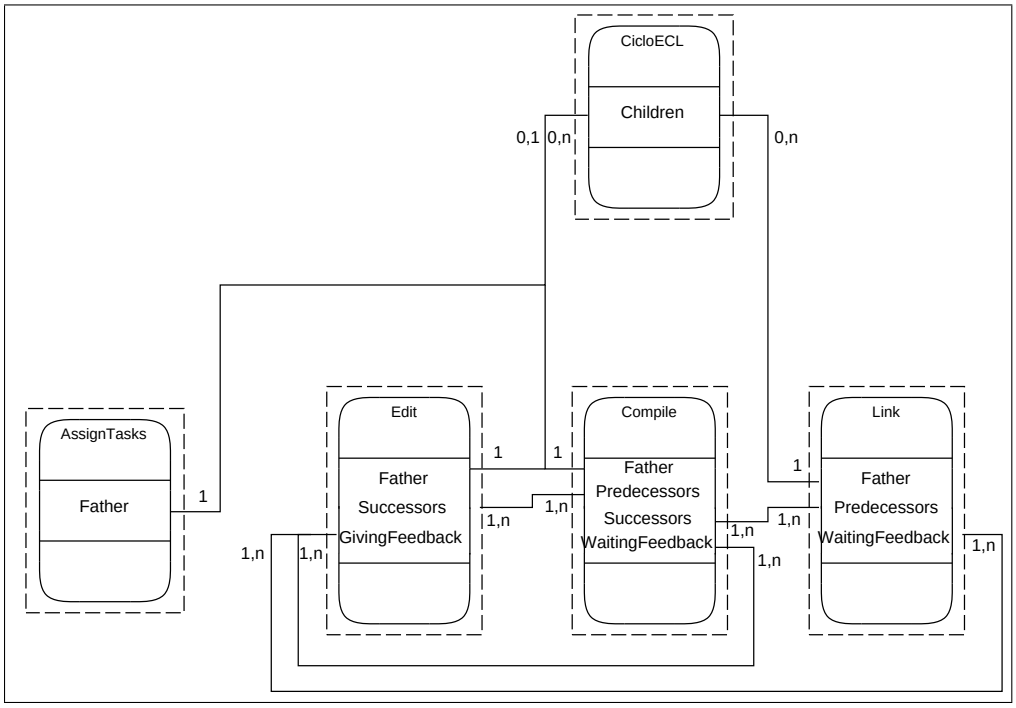

Figura 9.3. Esempio di decomposizione in sottoattività

`AssignTasks` permette di assegnare un certo numero di persone ed un responsabile ad una attività. Quindi ogni attività che si decompone in sottoattività deve avere la possibilità di istanziare un oggetto della classe `AssignTasks` la cui esecuzione consente l'associazione delle persone alle attività. Il funzionamento dettagliato del meccanismo di assegnazione (sezione 9.2.2) sarà spiegato dopo aver parlato della gestione delle persone coinvolte nel processo (sezione 9.2.1).

9.1.2 Le connessioni di istanza tra sottoclassi di `Task`

Si può notare che nella figura 9.3 sono tracciati tre tipi di connessioni tra gli oggetti delle sottoclassi di `Task`.

Padre/figlio Questa connessione collega gli attributi `Father` e `Children` specificando le sottoattività in cui un'attività si decompone; la relazione individuata riguarda sia le istanze delle classi che le classi stesse. Ad esempio

a livello di modello si dice che le attività di programmazione (modellizzate dalle istanze della classe `CicloECL`) avranno come figlie attività di scrittura di sorgenti (modellizzate dalle istanze della classe `Edit`); questo implica di avere una relazione tra la classe `CicloECL` e la classe `Edit`. Quando però il modello del processo viene eseguito si rivela necessario mettere in relazione anche le istanze delle classi suddette; infatti ciò che interessa sapere è quali sono le sottoattività di scrittura di sorgente (istanze di `Edit`) da cui è costituita un'attività di programmazione (istanza di `CicloECL`).

La cardinalità della connessione definisce, come è ovvio, che ogni attività ha un solo padre, ma il numero dei figli è determinante nella scelta del meccanismo che guida la creazione delle istanze che modellizzano i figli. Se il numero di figli è precisato, allora quando siano verificate le precondizioni, deve essere creato un numero di istanze della clase in questione pari alla cardinalità della connessione. Se invece il numero espresso è variabile (ad esempio 0,n), significa che il numero di istanze da creare dipende dai dati in ingresso o in uscita alle attività figlie, oppure dal meccanismo di decomposizione dell'attività composita. Il meccanismo standard di decomposizione è spiegato nel dettaglio trattando l'implementazione del modello di processo nella sezione 10.2.

Successori/predecessori È tracciata tra gli attributi `Predecessors` e `Successors` e serve ad indicare l'ordine di esecuzione tra le attività. La connessione a livello classe permette di definire nel modello, per un certo tipo di attività, quali altre la devono precedere e quali la devono seguire. Al momento dell'esecuzione del modello di processo diviene necessaria la connessione tra le istanze delle classi per sapere quali oggetti di `Task` ne precedono o seguono un altro. L'informazione data dalla connessione di livello classe viene sfruttata soprattutto nella istanziazione delle nuove attività (sezione 10.2) mentre quella data dalla connessione di livello istanza è utile per la coordinazione delle attività (sezione 10.1).

Feedback Collega gli attributi `WaitingFeedback` e `GivingFeedback` individuando le sottoclassi di `Task` a cui un'attività deve notificare il proprio fallimento e quelle il cui fallimento le è notificato. Come accade per i tipi di connessione precedentemente presentati, non è detto che la conoscenza debba essere reciproca; ad esempio ad un'attività interessa conoscere le attività

che deve notificare e non quelle da cui è notificata, ma questi sono problemi rimandati alla fase di implementazione, ed a livello di progettazione si considera che la connessione sia bidirezionale.

9.1.3 Il diagramma di stato degli oggetti di `Task`

L'attributo di maggior rilievo della classe `Task` è `State` il cui contenuto definisce lo stato in cui si trova l'attività modellizzata dall'oggetto. Il cambiamento di stato avviene a causa dell'esecuzione dei metodi secondo le modalità rappresentate dal diagramma di stato dell'oggetto `Task` riportato nella figura 9.4 e spiegate nel dettaglio nella sezione 10.5.

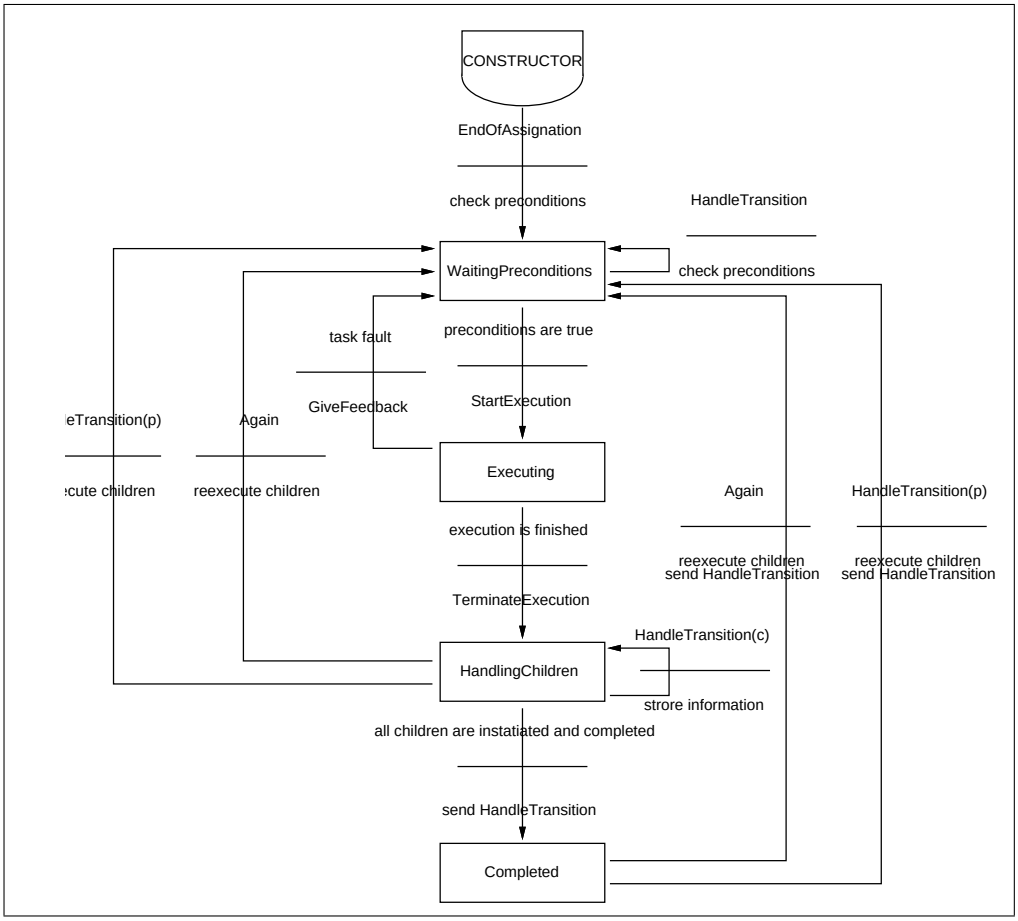

Figura 9.4. Diagramma degli stati degli oggetti di `Task`

Quando un'istanza di una sottoclasse di `Task` è creata entra nello stato `Constructor`. Nell'attuazione del modello di processo si utilizza l'*istanziazione incrementale* delle attività. In breve le attività sono istanziate solamente quando le condizioni che ne vincolano l'esecuzione sono tutte soddisfatte, cioè quando le attività che ne devono precedere l'esecuzione sono terminate. In realtà la terminazione dei predecessori non è l'unico fatto che vincola l'esecuzione di un'attività, ma devono verificarsi anche altre condizioni quali, ad esempio, l'avvenuta assegnazione di un responsabile.

Allora la nuova istanza permane in questo stato fino a che non riceve un messaggio di `EndOfAssignation` e quindi passa nello stato `WaitingPreconditions` in cui permane fino a che tutte le sue precondizioni non siano state soddisfatte. Infatti durante il tempo passato tra la creazione dell'istanza e l'avvenuta assegazione delle persone, le precondizioni potrebbero non essere più soddisfatte. Inoltre in questo modo l'esecuzione può essere abilitata da condizioni specifiche dell'attività in questione e non solo da quelle standard contemplate nella procedura di istanziazione dei figli definita per la classe `Task`.

Quando è possibile cominciare l'esecuzione vine invocato il servizio `StartExecution` che porta l'oggetto nello stato `Executing` e quindi realizza le operazioni specifiche dell'attività in questione e gestisce la terminazione e l'istanziazione dei figli.

L'esecuzione può venire più volte sospesa e ripresa, ma l'attività resta pur sempre nello stato `Executing` perché la sua sospensione riguarda solamente l'interazione con l'utente. Questa è una decisione presa nella realizzazione di S^3 perché non richiede differenti approcci, ma può essere riesaminata e variata nel caso si debba affrontare la modellizzazione di processi più complessi.

L'esecuzione può terminare con un fallimento nel cui caso devono essere notificate le varie attività collegate dalla connessione `WaitingFeedback` (a livello istanza). Questo compito è assolto dal servizio `GiveFeedback` che, dopo avere invocato il metodo `Again` delle istanze di `Task` interessate alla notifica, fa passare l'oggetto nello stato `WaitingPreconditions` (il meccanismo della notifica dei fallimenti è trattato più nel dettaglio nella sezione 10.3).

Se invece l'esecuzione termina con successo, viene invocato il servizio `TerminateExecution` che fa passare l'oggetto nello stato `HandlingChildren` in cui l'attività attende fino a che tutti i suoi figli sono terminati gestendo la comunicazione con essi e l'istanziazione di altri figli.

Quando tutti i figli sono terminati l'attività passa nello stato `Completed`

in cui è conclusa, ma da cui può tornare ad essere eseguita a causa del fallimento di altre attività.

La riesecuzione di un'attività può essere causata dalla ricezione di un messaggio `Again` dovuto al fallimento di un oggetto `Task` direttamente collegato dalla connessione `GivingFeedback`. Però può anche dovuta alla ricezione di un messaggio `HandleTransition` che indica che uno dei predecessori è stato rieseguito. In ogni caso questi meccanismi sono descritti nel dettaglio nelle sezioni 10.1 e 10.3.

9.2 Il soggetto `Role`

La classe `Role` non presenta attributi o servizi di particolare rilievo in quanto essa è utilizzata per assegnare ad ogni attività le persone adatte a portarla a termine. Nella figura 9.5 sono riportate le classi che modellizzano i ruoli che intervengono generalmente in un processo software. Quando nella modellizzazione di uno specifico processo si riveli necessario avere a disposizione altri ruoli è sufficiente creare altre sottoclassi di `Role` o di sue specializzazioni.

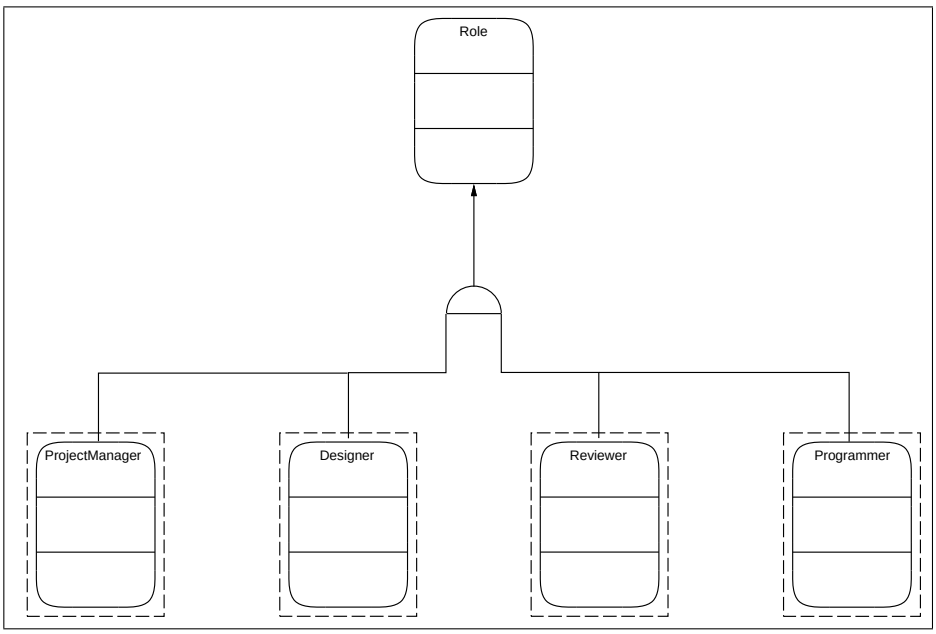

Figura 9.5. Il soggetto `Role`

Ogni attività ha un *responsabile* che deve interagire con il sistema fornendo le informazioni che questo non può produrre e che sono indispensabili per portare avanti il processo. I compiti del responsabile dipendono ovviamente dalla particolare attività che si sta considerando e possono essere strettamente legati alla generazione del prodotto del processo (realizzazione di un progetto, scrittura di codice), oppure allo svolgimento del processo stesso (stabilire quali attività debbano essere rieseguite per effetto di un fallimento).

Il responsabile di un'attività è individuato dalla connessione che lega l'attributo `Responsible` dell'oggetto della sottoclasse di `Task` che modellizza attività in questione, e l'attributo `CompetenceTasks` dell'oggetto di `Person` che modellizza la persona in questione. Questa connessione è definita tra le classi `Task` e `Person` come mostrato nella figura 8.1 ed è significativa solamente a livello delle istanze connesse. Come è evidenziato dalla cardinalità ogni attività ha un solo responsabile, ma una persona può essere responsabile di più attività.

Il responsabile di un'attività dovrà avere una certa competenza e determinate conoscenze che sono individuate dal ruolo che questo deve ricoprire. Si rivela allora necessario disporre di un mezzo per specificare il ruolo che deve essere ricoperto dal responsabile di qualsiasi attivtà. A tale scopo si ha una relazione tra `Task` e la sottoclasse di `Role` che modellizza il ruolo del responsabile; questa corrispondenza viene espressa nel modello dalla connessione definita a livello generale collegando l'attributo `ResponsibleRole` di `Task` all'attributo `CompetenceTasks` di `Role`.

Questa connessione non collega istanze delle due classi, ma ha significato solamente a livello delle classi perché il ruolo del responsabile è un'informazione che vale per tutte le attività di un certo tipo; ciò è evidenziato dal fatto che la cardinalità della 'instance connection' è zero. Perché la connessione abbia significato, deve essere ridefinita tra le sottoclassi di `Task` e quelle di `Role` in modo da dichiarare, mediante il modello, le competenze che devono essere proprie dei responsabili delle varie attività coinvolte nel processo.

Un caso particolare è quello in cui si volgia specificare che il responsabile di un'attività deve essere lo stesso dell'attività che l'ha generata decomponendosi. Si conviene di modellizzare ciò non specificando la connessione `ResponsibleRole/CompetenceTasks` tra le sottoclassi di `Task` e `Role` interessate. Questo implica che il meccanismo di istanziazione ed assegnazione dei responsabili alle sottoattività implementato nel padre, tenga conto di

questa convenzione associando la stessa persona responsabile per il padre, anche al figlio. Ciò evidenzia il fatto che per la modellizzazione dei processi software si deve avere a disposizione un ambiente che permetta di realizzare in modo coerente il progetto e l'implementazione del modello perché aspetti propri del progetto possobo basarsi su altri tipici dell'implementazione.

Sebbene le connessioni `Responsible/CompetenceTasks` e `Responsible-Role/CompetenceTasks` siano utilizzate, l'una per connettere istanze e l'atra classi, esse vengono entrambe rappresentate mediante lo stesso simbolo delle 'instance connection'. Conformemente a quanto detto nella sezione 6.5, ciascuna di esse rappresenta una relazione sia tra le classi, che tra le istanze, anche se in alcuni casi (come quello appena illustrato) una delle due può essere priva di significato e pertanto inutile.

9.2.1 Gestione delle persone e dei ruoli

Il problema di specificare i ruoli ricoperti dalle persone e di assegnare ad ogni attività il proprio responsabile è stato risolto utilizzando un meccanismo di assegnazione gerarchico delle persone alle attività.

Il modello delle attività che intervengono nel processo è gerarchico, cioè le attività si possono decomporre in sottoattività (normalmente più semplici). Questa decomposizione gerarchica è utilizzata per gestire in modo piuttosto semplice la coordinazione delle attività (sezione 10.1), la loro riesecuzione a causa di fallimenti (sezione 10.3) ed anche l'assegnazione delle persone in modo flessibile ed affidandola a diversi utenti.

Nella sezione 8.8 si è detto che ad un'attività è assegnata una persona in qualità di responsabile, ed un insieme di persone che sono a disposizione dell'attività. Per ognuna delle persone di questo insieme è necessario specificare il ruolo ricoperto, cioè tra attività e persona deve intercorrere una relazione che, oltre a connettere le due entità, identifichi il ruolo interessato. Per esprimere questo collegamento nel progetto si possono individuare due possibili soluzioni.

- Si utilizza una relazione binaria tra persona ed attività che ha in sé l'identificazione del ruolo. Questo può essere fatto in diversi modi uno dei quali è l'utilizzo di una classe le cui istanze realizzano la relazione portando in sé l'informazione del ruolo.

- Si utilizza una *relazione ternaria*: si crea un oggetto per ogni ruolo e lo

si collega mediante tale relazione all'attività, alla persona ed al ruolo ricoperto.

Nel progetto di S^3 si è seguito il primo approccio utilizzando come classe per realizzare la relazione proprio `Role`. L'informazione del ruolo ricoperto non viene da un attributo di una istanza, ma dalla sottoclasse di `Role` cui appartiene l'istanza.

Connessioni tra le sottoclassi di `Role` e le loro istanze

L'utilizzo delle sottoclassi di `Role` pare un modo efficiente, e soprattutto omogeneo con la tecnica di PM proposta, per esprimere a livello di modello il ruolo che deve essere ricoperto dal responsabile delle varie attività. Viene immediato utilizzare le istanze delle sottoclassi di `Role` per la realizzazione della suddetta relazione ternaria mantenendo uniformità con il resto del modello perché l'informazione del ruolo ricoperto resta sempre a livello di classe e sfruttando le istanze di una classe che diversamente sarebbe astratta.

Allora, per esprimere il fatto che le persone `P1` e `P2` sono a disposizione dell'attività `A` nel ruolo `R`:

- si crea una nuova istanza `I` della classe `R` (sottoclasse di `Role`);
- si connette `I` ad `A` mediante la connessione che unisce l'attributo `AssignedRoles` di `Task` e `AssignedTask` di `Role` (mostrata nella figura 8.1);
- si connette `I` a `P1` e `P2` mediante la connessione tra l'attributo `AssignedPersons` di `Role` e `AssignedRoles` di `Person` (mostrata nella figura 8.1).

Dunque si ha un'istanza di una sottoclasse di `Role` per ogni attività per cui siano a disposizione una o più persone nel ruolo in questione. Questa corrispondenza è evidenziata dalla cardinalità della connessione `AssignedRoles/AssignedTask` la quale afferma che ad ogni istanza di `Role` è associata una ed una sola istanza di `Task`, mentre ad ogni istanza di `Task` può essere associato un qualunque numero di istanze di `Role`.

Una persona può ricoprire più di un ruolo per la stessa attività e per questo nella connessione `AssignedRoles/AssignedPersons` la cardinalità afferma che un qualunque numero di istanze delle due classi può essere collegato.

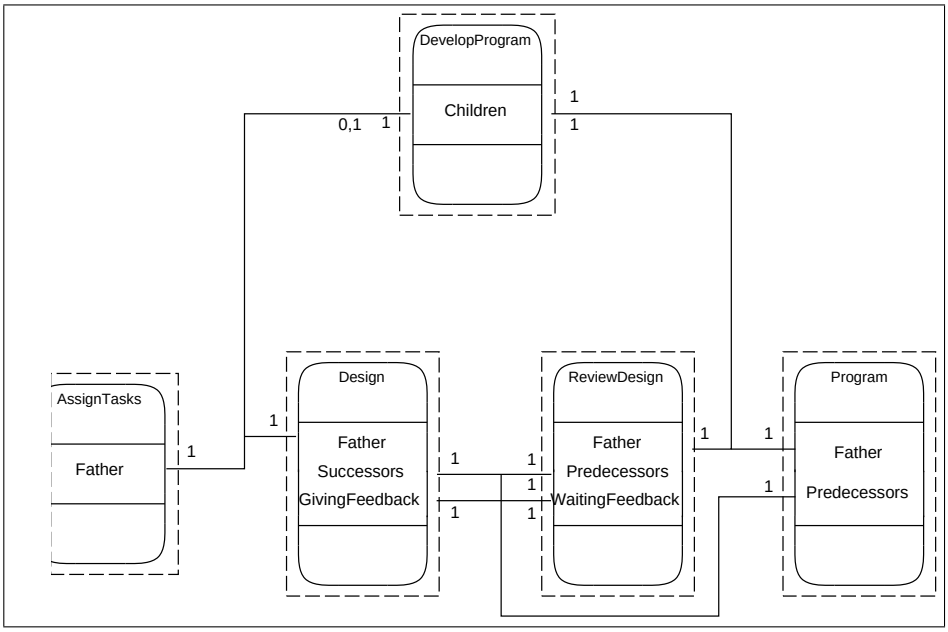

Figura 9.6. Esempio della parte alta della gerarchia di decomposizione di un modello di processo

Allora al momento dell'istanziazione del modello di processo per ogni persona si definiscono i ruoli che questa ricopre nel processo in questione effettuando i collegamenti descritti in precedenza con l'attività di più alto livello. Nella figura 9.6 è riportato la parte alta di una gerarchia di decomposizione di attività. Tutte le persone che intervengono nell'esecuzione del processo software da cui questa gerarchia è stata estrapolata, sono assegnate all'istanza di `DevelopProgram` che, in quanto attività di più alto livello, rappresenta l'intero processo software. Quando `DevelopProgram` si decompone nelle sue sottoattività, le persone vengono assegnate con i loro ruoli alle sottoattività secondo il meccanismo descritto nella sezione 9.2.2.

Un esempio dell'utilizzo di `Role`, `Task`, `Person` e le relazioni tra di esse e le loro istanze è mostrato nella figura 9.7. In questa figura si è utilizzata una notazione che permetta di rappresentare sia il livello delle istanze che quello delle classi; le classi sono state rappresentate con dei quadrati mentre le istanze con dei cerchi. Le linee che collegano quadrati e cerchi indicano che il cerchio rappresenta un'istanza della classe rappresentata dal quadrato collegato; queste attraversano sempre la riga che separa il livello classe da

quello delle istanze. Le linee che sono completamente tracciate nel livello classe o in quello delle istanze rappresentano relazioni tra le classi o tra le istanze rispettivamente.

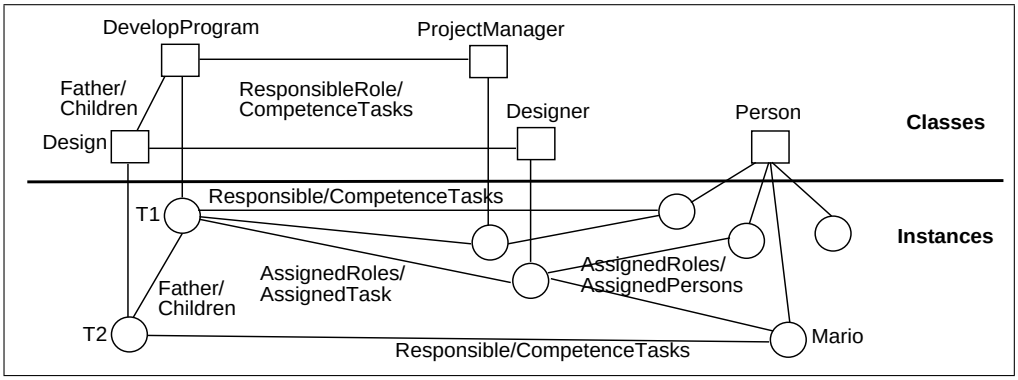

Figura 9.7. Un esempio di gestione di persone e ruoli

In questo esempio si può notare che le attività di tipo `Design` devono avere come resonsabile un `Designer` e `T2` è collegata dalla connessione `ResponsibleRole/CompetenceTasks` a `Mario`. Questo, a sua volta, è una delle persone assegnate a `T1`, padre di `T2`, in qualità di `Designer`; infatti è connesso mediante `AssignedPersons/AssignedRoles` ad un'istanza di `Designer` che è collegata a `T1` da `AssignedRoles/AssignedTask`.

9.2.2 Assegnazione delle persone alle attività

Le operazioni di assegnazione di persone ad attività sono gestite dal metodo definito nella classe `Task` per la decomposizione in sottoattvità. Quando l'assegnazione non può essere fatta in modo automatico, perché è necessario scegliere tra un certo numero di persone candidate, le decisioni sono prese dal responsabile dell'attività che si decompone e l'interazione con questa persona è gestita da una particolare attività definita dal sistema che è modellizzata dalle istanze della classe `AssignTasks`.

Come si può notare nella figura 9.3 e nella figura 9.6, qualsiasi attività composita deve essere collegata mediante la connessione `Father/Children` ad `AssignTasks`. La cardinalità esprime la possibilità di non avere figli di questo tipo o di averne al più uno perché non è detto che l'interazione con l'utente sia indispensabile.

La scelta di utilizzare una sottoattività per l'assegnazione interattiva delle persone ai figli, nasce dalla volontà di mantenere omogeneità nel sistema. Infatti l'interazione dell'utente per scegliere alcuni dei candidati, si può vedere come una qualsiasi altra attività del processo che rende noto al proprio responsabile il fatto che necessita del suo intervento per poter essere portata a termine. Ovviamente è possibile risolvere la questione in modo differente, ad esempio facendo gestire l'interazione con l'utente direttamente dall'attività composita.

`AssignTasks` non ha una connessione `ResponsibleRole/Competence-Tasks` perché il suo responsabile è lo stesso dell'attività che la ha istanziata. Inoltre, anche se non è mostrata nella figura 11.3, vi è una connessione `To-Assign/Assigner` con la classe `Task`. Quando un'attività composita crea le istanze delle proprie figlie, se qualcuna di questa necessita dell'interazione dell'utente per l'assegnazione del responsabile o delle persone a disposizione, l'attività composita istanzia anche una figlia `AssignTasks` che connette alle attività da assegnare mediante `ToAssign/Assigner`. Se esiste già una figlia di tipo `AssignTasks`, le nuove attività vengono collegate a questa che, se si trova nello stato `Completed`, viene riattivata.

Quando il responsabile di `AssignTasks` ha operato le scelte mediante un'opportuna interfaccia, `AssignTasks` comunica le assegnazioni alle attività che sono collegate mediante `ToAssign/Assigner` ed elimina i collegamenti. Se non restano altre attività cui assegnare persone, e quindi nessun collegamento `ToAssign/Assigner` creato mentre era in esecuzione, `AssignTasks` si porta in stato `Completed`. Diversamente ricomincia la propria esecuzione aspettando che l'utente si renda nuovamente disponibile per fare altre assegnazioni.

9.3 Il soggetto `Data`

Le classi di questo soggetto derivano per ereditarietà dalla classe `Data` secondo la gerarchia mostrata nella figura 9.8 e rappresentano qualsiasi entità che possa essere presa in ingresso o prodotta da un'attività. A questo scopo sono definite tra `Data` e `Task` le connessioni `Inputs/Users` e `Outputs/Productors` che collegano le attività ai loro ingressi e prodotti rispettivamente. Quando si realizza un modello di processo si creano sottoclassi di `Task` e `Data`; tra queste sottoclassi vanno ridefinite le suddette connessioni perché portano informazione a livello di classe, oltre che a quello

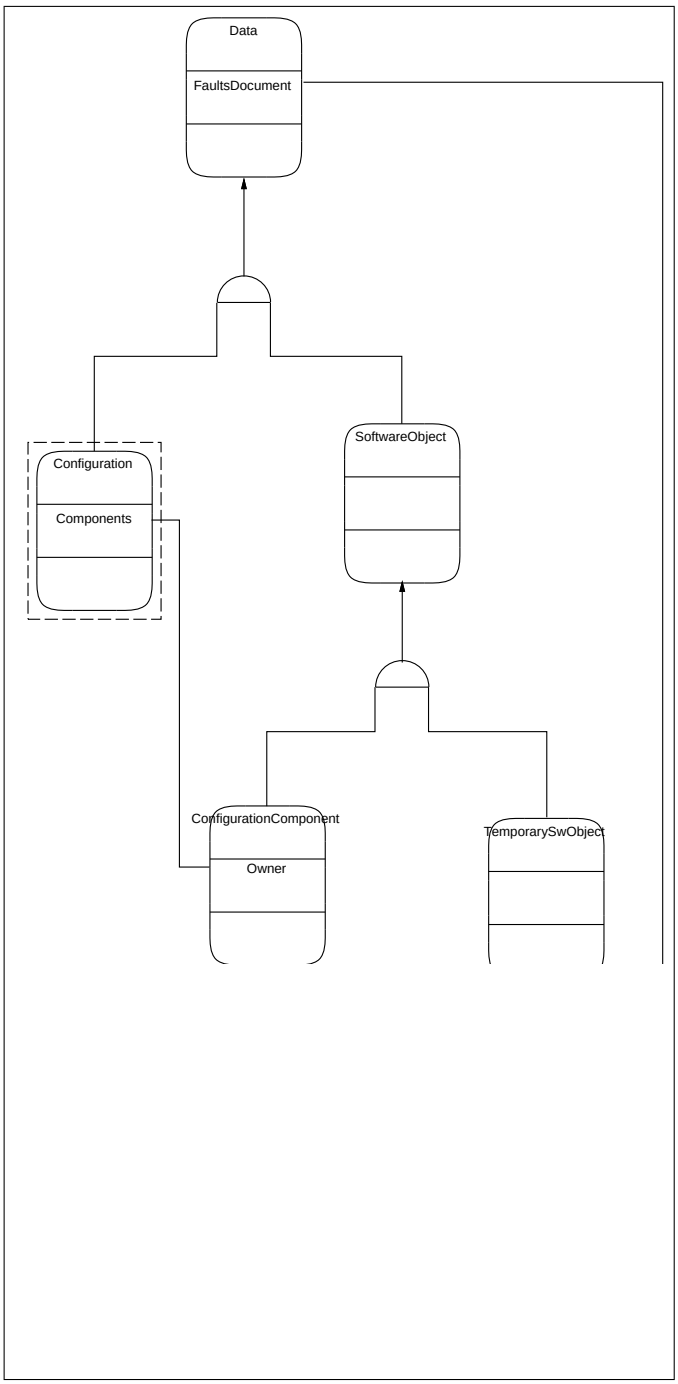

Figura 9.8. Le classi predefinite del soggetto `Data`

delle istanze. Le connessioni `Inputs/Users` e `Outputs/Productors` sono di

notevole importanza nella gestione degli strumenti automatici utilizzati nel processo (sezione 9.4.1) e nell'istanziazione delle sottoattività (sezione 10.2).

Nella figura 8.1 oltre a queste connessioni è riportata `Creator/Created` tra `Data` e `Tool` che identifica lo strumento automatico utilizzato per creare e manipolare il dato rappresentato dall'istanza di `Data`. Anche questa connessione ha significato utile solo qualora sia ridefinita tra sottoclassi di `Data` e `Tool` perché deve fornire un'informazione a livello classe oltre che istanza; il suo utilizzo è spiegato trattando della gestione degli strumenti automatici nel processo nella sezione 9.4.1.

9.3.1 Le sottoclassi predefinite di `Data`

Un caso particolare di ingresso o di prodotto di un'attività è l'oggetto software, generalmente un 'file', che è modellizzato dalle istanze della classe `SoftwareObject` che è una diretta specializzazione di `Data`.

Da `SoftwareObject` sono derivate per specializzazione una serie di classi che modellizzano ognuna un differente tipo di oggetto software che deve essere prodotto o manipolato da una qualche attività. `SoftwareObject` non ha istanze e da essa derivano altre due sottoclassi astratte:

- `ConfigurationComponent` che modellizza gli oggetti software che influenzano il prodotto finale e quindi devono essere parte della sua configurazione;
- `TemporrarySwObject` che modellizza gli oggetti software che vengono create temporaneamente e, dopo il loro utilizzo, eliminati.

Diretta specializzazione di `Data` è `Configuration` che serve a modellizzare la configurazione del prodotto software che si vuole sviluppare con il processo. Infatti i prodotti delle varie attività sono tra loro correlati e sono finalizzati alla produzione di uno o più oggetti software che costituiscono il prodotto finale del processo ('*deliverable*'). Tutti questi oggetti software, istanze di sottoclassi di `ConfigurationComponent`, vanno raggruppati ed anche mantenuti consistenti tra di loro a mano a mano che vengono modificati dal processo.

Assicurare la consistenza è un problema molto complesso di cui si occupano le discipline della *Gestione delle Configurazioni* ('*Configuration Management*') e della *Gestione delle Versioni* ('*Version Management*') le cui

funzioni si suppongono assolte da uno strato sottostante al sistema per la modellizzazione dei processi software.

Il raggruppamento avviene creando un oggetto `Configuration` a cui sono connesse le varie istanze delle sottoclassi di `ConfigurationComponent` che sono prodotte durante il processo mediante la connessione tra l'attributo `Components` di `Configuration` e l'attributo `Owner` di `ConfigurationComponent`. Questa connessione va ridefinita per le sottoclassi di `ConfigurationComponent` perché fornisce informazioni utili a livello classe per la definizione della configurazione del prodotto come spiegato nella sezione 9.3.6.

Ci sono oggetti software che sono istanze di sottoclassi di `TemporarySwObject`, che vengono creati per scopi unicamente relativi al processo e che non hanno nulla a che vedere con il prodotto finale. Tra le classi predefinite in S^3 compare la specializzazione di `TemporarySwObject` `FeedbackDocument` che rappresenta i documenti utilizzati nella gestione del fallimento delle attività come spiegato nella sezione 10.3.

Gli oggetti software modellizzati da istanze di sottoclassi di `TermporarySwObject` non vengono inclusi nella configurazione del prodotto finale, pur avendo anch'essi delle relazioni con gli oggetti software che costituiscono questa configurazione. Infatti il fallimento di un'attività è spesso dovuto al suo ingresso a cui deve quindi essere associato il rapporto del fallimento. Questo è fatto mediante la connessione `ReferredTo/FaultsDocument` tra `FeedbackDocument` e `Data` che viene utilizzata a livello istanza.

9.3.2 `ConfigurationComponent` ed il versionamento

Nella sezione 8.9 sono esposti i principi su cui ci si basa per trattare la gestione delle versioni in S^3; in questa sezione è trattato il modo in cui questa è stata integrata con la tecnica di modellizzazione proposta.

Si è realizzata la classe `ConfigurationComponent` che rappresenta un oggetto software, tipicamente un 'file', che è parte di una configurazione, ma soprattutto che è soggetto a versionamento. Quindi si formulano le seguenti ipotesi che stanno alla base del metodo di modellizzazione presentato in questa trattazione:

1. ogni volta che un'attività produce un dato modellizato da un'istanza della classe `ConfigurationComponent`, esso è memorizzato come nuova

versione mediante l'invocazione di un apposito metodo della classe in questione;

2. ogni volta che un'attività deve utilizzare un dato invoca un metodo dell'oggetto che lo modellizza per ottenere tale dato e le viene fornita la versione più recente di tale dato.

L'aver incapsulato le questioni riguardanti la gestione delle versioni nella classe `ConfigurationComponent` permette, sfruttando uno dei più potenti meccanismi messi a disposizione dall'orientamento agli oggetti, di trascurare i dettagli inerenti all'implmentazione del versionamento ed anche di supporre che la gestione di quest'ultimo sia affidata ad un supporto sottostante all'ambiente di attuazione del processo. Questo può essere assunto in generale, ma nella realizzazione di S^3 che, essendo un ambiente per la simulazione, si sono trascurati completamente gli aspetti legati alla produzione dei dati da parte delle attività. Questo non rappresenta comunque un limite sui risultati ottenuti dalla simulazione proprio grazie al fatto che, con le ipotesi fatte, la gestione della memorizzazione dei dati e delle loro versioni è indipendente dal processo in sé e dalla sua esecuzione.

9.3.3 Le sottoclassi di `ConfigurationComponent`

Quando si realizza un modello di processo software, le sottoclassi di `ConfigurationComponent` servono per esprimere, a livello di modello, la struttura del prodotto, cioè di che genere devono essere i vari componenti della configurazione. Queste classi e le loro connessioni costituiscono il *modello dei dati* di un modello di processo. Le istanze delle sottoclassi di `ConfigurationComponent` vengono create durante il processo e rappresentano i vari componenti della configurazione del prodotto. Nella figura 9.9 è riportata una parte di un modello di processo software che permette di descrivere una struttura piuttosto generale di prodotto. Questo modello dei dati, sebbene non faccia parte di S^3 è molto generale e può essere utilizzato, magari parzialmente, nella modellizzazione di processo software.

- `RequirementDocument` modellizza i documenti dei requisiti per il prodotto finale del processo. Ovviamente questi documenti non sono il prodotto di una delle attività, ma devono essere forniti al momento dell'istanziazione del processo. Sarà compito delle fasi iniziali del processo creare le opportune istanze di questa classe e collegarle ai 'file'

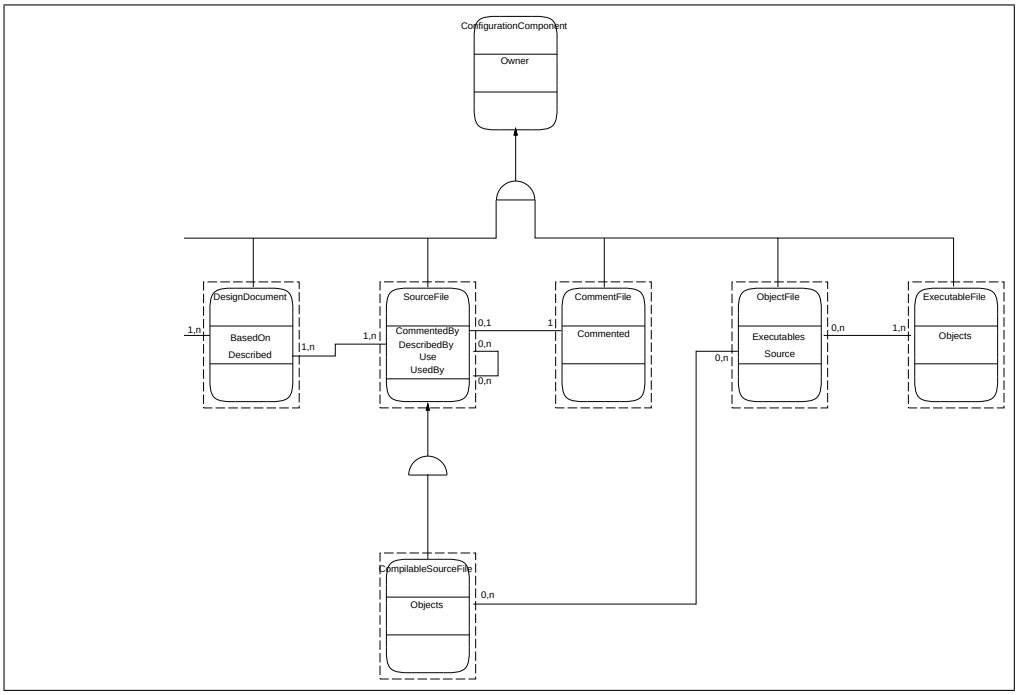

Figura 9.9. Esempio di modello di prodotto

che effettivamente contengono i requisiti per renderli disponibili al processo.

- `DesignDocument` rappresenta i documenti contenenti il progetto che sono il prodotto dell'attività `Design`.

- `SourceFile` modellizza i 'file' che contengono il codice e che vengono compilati. Si tratta soprattutto di 'file' che vengono inclusi in altri, come ad esempio quelli di intestazione ('header'). Questi dati vengono prodotti dalle attività di `Edit` e sono forniti in ingrasso alle attività di compilazione. Se si suppone che il prodotto del processo debba essere ottenuto servendosi del linguaggio C, le istanze di questa classe saranno i 'file' con estensione *.h* o *.c* che vengono inclusi dai sorgenti descritti dalla classe trattata di seguito.

- `CompilableSourceFile` è una specializzazione della classe precedente le cui istanze rappresentano i 'file' sorgente che vengono normalmente passati come argomento al compilatore che li utilizza per produrre il

codice oggetto. Questi sorgenti fanno uso di quelli modellizzati dalla classe `SourceFile` a cui vengono uniti in fase di compilazione.

- `CommentFile` è un documento di commento ad un qualsiasi sorgente (istanza di `SourceFile` o sua sottoclasse); è prodotto da un'attività di `Edit` e non è utilizzato da nessuna attività, ma serve come documentazione al sorgente cui è legato.

- `ObjectFile` modellizza i 'file' contenenti il codice oggetto prodotto dai compilatori (`cc` nel caso dell'esempio di processo) e quindi dall'attività di compilazione `Compile`.

- `ExecutableFile` è l'eseguibile ottenuto dai 'file' oggetto mediante l'attività `Link` e generalmente costituisce il prodotto finale ('deliverable') del processo.

Le relazioni tra i dati e le attività che li producono sono spiegate più nel dettaglio nella sezione seguente.

9.3.4 Le connessioni tra `Data` e `Task`

Tra la classe `Data` e la classe `Task` vengono definite due connessioni che sono riportate nella figura 8.1:

- `Inputs/Users` mette in relazione i dati all'attività che ne fanno uso, senza modificarli;

- `Outputs/Productors` mette in relazione i dati con le attività che li manipolano o li producono.

Entrambe queste connessioni hanno significato sia a livello classe che a livello istanza.

Livello classe

Come accade tutte le volte che si deve sfruttare una connessione a livello classe, sebbene sia definita tra le classi più generali connesse, essa va specificata tra le sottoclassi. Un esempio è riportato nella figura 9.10 in cui la connessione `Outputs/Productors` è ridefinita tra le classi `Design` e `DesignDocument`

per indicare che le attività di progettazione agiscono su oggetti software modellizzati da istanze della classe `DesignDocument`, o sue sottoclassi. Nel modello di un processo realizzato con la tecnica proposta, è indispensabile specificare gli ingressi e le uscite di tutte le attività.

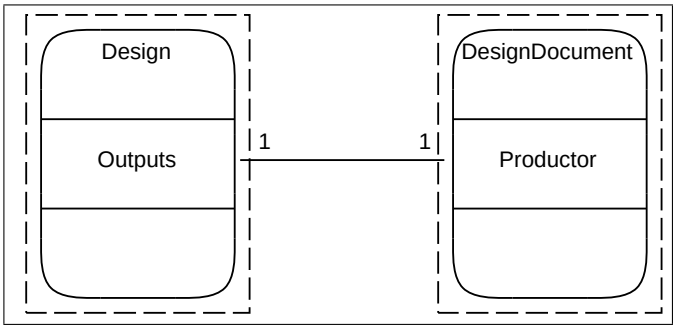

Figura 9.10. Esempio di connessione `Ouputs/Productors`

Le informazioni date dalle connessioni di livello classe sono sfruttate dal padre delle attività in situazioni differenti:

- quando gli occorre sapere i tipi di strumenti automatici che dovrà utilizzare in modo che gli vengano associati secondo le modalità espresse nella sezione 9.4.1;
- quando durante la decomposizione deve sapere che tipo di dati producono le sottoattività, per capire se sia necessario creare una nuova sottoattività oppure per individuarne i predecessori secondo quanto illustrato nella sezione 13.4;
- quando, dopo avere istanziato una sottoattività, le deve collegare gli oggetti che effettivamente utilizza in ingresso e quelli che produce o modifica.

Livello istanza

Dopo che un'attività è stata istanziata, essa viene connessa ai dati che le occorrono mediante le connessioni `Input/Users` ed tt Ouput/Productors di livello istanza. A questo punto l'attività può accedere tramite queste connessioni agli oggetto `Data` che la interessano ed non utilizza mai le corrispondenti relazioni a livello classe.

L'individuazione delle istanze di `Data` da connettere, note le loro classi, è tutt'altro che banale. A questo scopo viene utilizzato un algoritmo che, a partire dall'oggetto che deve essere prodotto, cerca tutti gli altri, de collegare come ingresso e come uscita, tramite le connessioni tra le istanze di `Data`. Tale ricerca avviene solamente tra gli oggetti in qualche modo correlati agli ingressi ed alle uscite dell'attività che si sta decomponendo. La spiegazione dettagliata si trova nella sezione 13.4.5 in cui è riportata l'implementazione del metodo che esegue questa operazione.

9.3.5 Le connessioni di istanza nel soggetto `Data`

Si è già detto nella sezione 9.3.1 che la connessione `Components/Owner` permette di idenitificare i diversi componenti di una configurazione.

`ReferredTo/FaultsDocument` collega un'istanza di `FeedbackDocument` ad una di una sottoclasse di `Data`. Quando un'attività fallisce e genera una notifica del fallimento, essa fornisce anche un elenco dei motivi che hanno impedito la corretta terminazione. Tali motivi saranno ovviamente legati ai dati che l'attività in questione ha in ingresso e così il documento contenente i problemi riscontrati (modellizzato da un'istanza di `FeedbackDocument`) viene collegato ad essi.

Nella figura 9.9 si è riportato un esempio di modello dei dati. Nel realizzare un tale modello non sono importanti solo le classi, ma anche le connessioni tra esse che permettono di stabilire le opportune relazioni tra gli oggetti software che compongono la configurazione. Poiché questo modelo dei dati è molto generale e direttamente utilizzabile per la modellizzazione di processi software, può essere utile spiegare nel dettaglio le connessioni che sono definite in esso.

- `MetBy/BasedOn` tra `RequirementDocument` e `DesignDocument` lega ogni progetto ai documenti contenenti i requisiti su cui si è basata la progettazione. Le cardinalità agli estremi della connessione (entrambe 1,n) indicano che un documento di requisiti deve dare origine ad uno o più documenti di progetto e che un documento di progetto deve essere stato realizzato secondo i requisiti contenuti in uno o più documenti.

- `Described/DescribedBy` collega ogni istanza di `SourceFile` alle istanze di `DesignDocument` che rappresentano i documenti contenenti il progetto di tali sorgenti. La cardinalità della connessione evidenzia il fatto

che ogni documento di progetto deve essere implementato in almeno un sorgente e che ogni sorgente deve avere un progetto corrispondente in almeno un'istanza di `DesignDocument`.

- `Commented/CommentedBy` mette in relazione un sorgente con il proprio documento di commento. La cardinalità come espressa nella figura 9.9 impone che ogni 'file' di commento descriva un unico sorgente, ma si possono fare scelte, a livello di modello, di tipo differente. Questo modello dei dati non impone che ogni 'file' sorgente abbia un documento di commento associato, ma è possibile fare questo usando cardinalità 1 invece che 0,1.

- `Use/UsedBy` collega alle istanze di `CompilableSourceFile`, o anche di `SourceFile` stesso, agli oggetti `SourceFile` che rappresentano i sorgenti inclusi durante la compilazione. Non è detto che un sorgente da compilare faccia delle inclusioni (cardinalità 0,n) ed inoltre è possibile che uno stesso `SourceFile` sia incluso in più 'file' compilabili. Il fatto che un'istanza di `SourceFile` possa non essere connessa ad oggetti `CompilableSourceFile` permette di avere una libreria di *'header'* e di *'includes'* alcuni dei quali possono anche non essere utilizzati.

- `Objects/Source` serve per collegare un 'file' oggetto all'istanza di `CompilableSourceFile` che è stata compilata per generalo. Dalla cardinalità della connessione si può notare che uno stesso sorgente può generare più di un 'file' oggetto nel caso in cui si utilizzino versioni diverse di compilatore o parametri di compilazione differenti.

- `Executables/Objects` mette in relazione ogni eseguibile con i 'file' oggetto che sono stati sottoposti all'azione del 'linker' per generarlo. Le cardinalità esprimono il fatto che lo stesso codice oggetto può essere riutilizzato per produrre più di un eseguibile.

9.3.6 La configurazione del prodotto

Un aspetto molto importante nella produzione del software è la definizione della configurazione del prodotto. Nella modellizzazione dei processi si vuole fornire una descrizione del processo per ottenere un prodotto software, ma non si può già definirne la configurazione; infatti normalmente essa è decisa

nel momento in cui viene fatta la progettazione del prodotto, o comunque durante l'attuazione del processo stesso.

Dunque il sottomodello dei dati non fa altro che stabilire la tipologia degli elementi che potranno essere utilizzati nella fase di progettazione per definire la configurazione del prodotto, e le relazioni che dovranno essere imposte tra questi elementi. Questo, nella terminologia propria dell'approccio ad oggetti si può esprimere dicendo che nel modello si definiscono delle classi le cui istanze si possono utilizare per definire la struttura del prodotto collegandole mediante le 'instance connection' dichiarate nel modello tra queste classi.

Quando il modello di processo viene eseguito, l'attività incaricata di definire la configurazione del prodotto finale (generalmente quella di progettazione), deve creare la configurazione interagendo con il proprio responsabile. La struttura del prodotto è però definita dal modello dei dati per cui la creazione della configurazione implica un'ispezione a livello classe per vedere, nel modello del processo, quali sono le sottoclassi di `ConfigurationComponent` e le relazioni tra di esse in modo da sapere di che *tipo* sia la struttura del prodotto. A questo punto, sapendo la tipologia delle entità che si possono utilizzare per costruire la configurazione, si creano le istanze delle varie sottoclassi di `ConfigurationComponent` che rappresentano gli oggetti sofware coinvolti e li si collega mediante le connesioni definite nel modello.

Nella figura 9.11 è mostrato un esempio di una parte della configurazione di un prodotto utilizzando lo stesso formalismo usato nella figura 9.7 che permette di rappresentare sia il livello delle istanze che quello delle classi. Questa configurazione è stata creata a partire dal modello dei dati presentato nella figura 9.9.

9.4 Il soggetto `Tool`

Contiene la gerarchia delle classi utilizzate per la modellizzazione degli strumenti automatici che sono usati delle attività per generare i loro prodotti. La classe `Tool`, che dà il nome al soggetto, è la radice della gerarhia e le sue specializzazioni sono utilizzate per modellizzare i vari tipi di strumenti automatici; ognuno, di essi è rappresentato da un'istanza di una sottoclasse di `Tool`.

La classe `Tool` è collegata alla classe `Data` dalla connessione `Created/Creator` mostrata nella figura 8.1. A livello di istanza essa permette di identificare per ogni dato lo strumento automatico con cui è stato creato

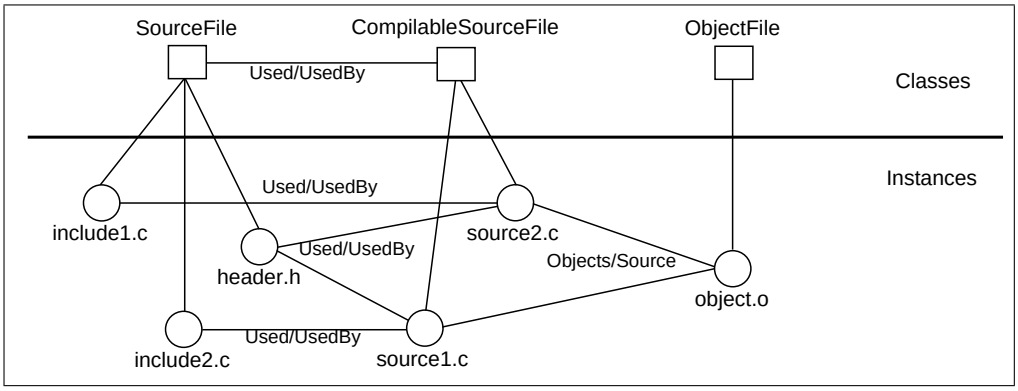

Figura 9.11. Un esempio di configurazione

e da cui può essere manipolato.

A livello classe la connessione `Created/Creator` permette di identificare il tipo di strumento automatico che è in grado di operare su di un certo tipo di dato. Per questo motivo tale connessione va ridefinita tra le opportune sottoclassi di `Data` e di `Tool` perché acquisti un significato utile. Quale sia lo strumento effettivamente utilizzato, cioè la connessione tra le istanze delle rispettive classi, dipende all'attività che crea il dato e dalle preferenze del responsabile come spiegato nella sezione 9.4.1; in ogni caso tale strumento sarà scelto tra quelli modellizzati dalle istanze della sottoclasse di `Tool` connessa alla sottoclasse di `Data` in questione.

9.4.1 Gestione degli strumenti automatici

Ogni attività ha a disposizione un insieme di strumenti automatici che è in grado di individuare grazie alla connessione tra l'attributo `AssociatedTools` della classe `Task` e l'attributo `AssociatedTasks` di `Tool` (figura 8.1).

Associazione degli strumenti alle attività

Quando un'attività `A` si decompone in sottoattività, ad ogni figlia `F` sono automaticamente associati tutti gli strumenti, a disposizione di `A`, che sono istanze di una sottoclasse di `Tool T` utile ad `F`. Cioè `T` deve essere connessa mediante `Creator/Created` ad almeno una sottoclasse di `Data` utilizzata come ingresso o prodotta in uscita da `F` o da una delle sue sottoattività.

Per restringere l'insieme di strumenti automatici a disposizione di un'attività occorre utilizzare un'istanza della classe `SelectTool`, predefinita in S^3; le modalità secondo cui tale clase deve essere inserita nel modello di processo ed i principi del suo funzionamento sono espressi nella sezione 9.4.2.

Quando un'attività è eseguita, nel caso le siano disponibili più strumenti dello stesso tipo, il responsabile può esprimere la propria preferenza riguardo quello da utilizzare. Tale preferenza è memorizzata mediante la connessione `ChosenTools/ChosenBy` tra `Person` e `Tool` e viene tenuta in conto nelle successive situazioni dello stesso tipo.

Meccanismo di scelta degli strumenti

Molte attività ad un certo punto della propria esecuzione devono lanciare gli strumenti automatici sui dati che hanno in ingresso e su quelli che devono produrre. Allora, l'oggetto di `Task` in questione per ogni dato controlla qual è lo strumento che agisce su di esso; se questo è già stato specificato, cioè il dato è già stato creato, lo utilizza per manipolare il dato in questione.

Se il dato non è ancora stato creato, e non è quindi connesso ad un'istanza di `Tool` mediante la connessione `Created/Creator`, l'attività deve scegliere uno strumento da utilizzare tra quelli che le sono stati associati. Nel fare questo essa deve seguire i passi sottoelencati per ogni dato utilizzato dall'attività o prodotto da essa.

1. Individua il tipo di strumento in grado di operare sul dato medinate la connessione `Created/Creator` di livello classe.

2. Controlla quanti strumenti del tipo in questione le sono associati mediante la connessione `AssociatedTools/AssociatedTasks`:

   - se ce ne è uno solo lo utilizza connettendolo mediante `Created/Creator` al dato in questione;
   - se ce n'è più di uno:
     - se il responsabile ha una preferenza (espressa dalla connessione `ChosenTools/ChosenBy`) per uno degli strumenti associati all'attività, viene utilizzato quest'ultimo;
     - se il responsabile non ha preferenze gli viene chiesto di scegliere uno tra gli strumenti a disposizione.

Nella figura 9.12 è riportato un esempio di connessioni tra sottoclassi di `Tool`, `Task`, `Data` e loro istanze realizzato secondo i principi descritti in questa sezione. L'attività `anEditing` deve produrre `interf.h` che, come si vede dalla connessione `Created/Creator` di livello classe, può essere manipolato da strumenti della classe `Editor`. L'unico strumento associato mediante `AssociatedTools/AssociatedTasks` a `anEditing` di questo tipo è `vi` e quindi lo usa collegandolo ad `interf.h` mediante la connessione `Created/Creator` di livello istanza.

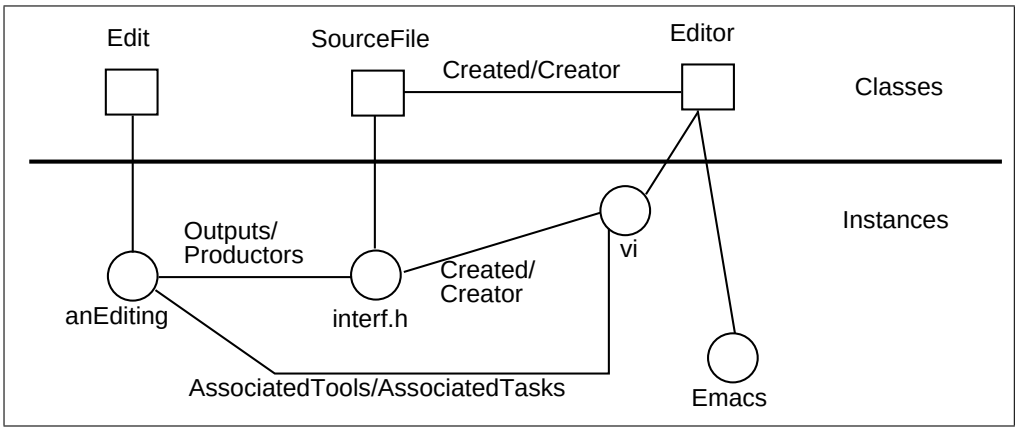

Figura 9.12. Esempio di utilizzo degli strumenti automatici

Si può facilemente dedurre dall'algoritmo presentato che per imporre l'utilizzo di uno strumento automatico è sufficiente fare in modo che esso sia l'unico disponibile del suo tipo.

Si può notare che, come l'assegnazione delle persone, anche la gestione degli strumenti automatici è fatta in modo gerarchico. Cioè per ogni attività sono disponibili solamente strumenti che sono associati a chi la ha generata per cui è possibile imporre l'uso di un particolare strumento a tutto un ramo della gerarchia delle attività, semplicemente imponendolo all'attività alla radice di questa gerarchia.

9.4.2 Funzionamento di `SelectTool`

Quando un'attività si decompone, alle attività generate sono automaticamente resi disponibili tutti gli strumenti associati al padre. In questo modo non sarebbe possibile imporre l'uso di un certo strumento per un'attività,

se all'intero processo è disponibile più di un 'tool' di quel tipo. Infatti tutti gli strumenti associati all'attività di più alto livello sono automaticamente disponibili alle sue sottoattività a mano a mano che sono istanziate, e quindi a tutto il processo.

Effetto di `SelectTool`

`SelectTool` è un'attività di sistema, cioè non caratteristica di un particolare processo, ma fornita da S^3 per realizzare meccanismi propri dell'ambiente, e serve ad imporre l'uso di un preciso strumento. La sua esecuzione consta nel mostare un'interfaccia all'utente che gli consente, per ogni tipo di strumento disponibile all'attività padre, di sceglierne uno. Ovviamente questa attività è utile solo in quei casi in cui più di uno strumento sia disponibile per ogni tipo. Vengono quindi mantenute per l'attività padre solo le connessioni `AssociatedTools/AssociatedTasks` verso gli strumenti scelti.

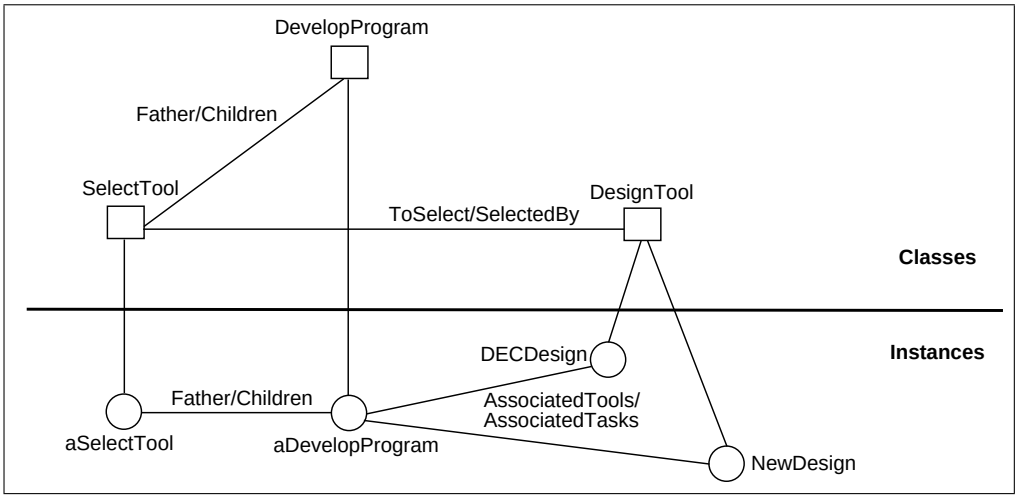

Figura 9.13. `AssociatedTools/AssociatedTasks` prima dell'esecuzione di `SelectTool`

Nella figura 9.13 è riportato una parte di un modello di processo (livello classe) ed un istante della sua attuazione (livello istanza) fissato prima dell'esecuzione dell'attività `aSelectTool`. Sono riportate le connessioni dall'attività `aDevelopProgram` (istanza di `DevelopProgram`) verso gli strumenti di sussidio alla progettazione (istanze di `DedignTool`) prima dell'esecuzione

dell'istanza di `SelectTool`. Dopo l'esecuzione di `aSelectTool` le connessioni divengono quelle mostrate nella figura 9.14.

L'imposizione di `DECdesign` come strumento di sussidio alla progettazione sarà fatta solo per quelle attività che sono istanziate dopo l'esecuzione di `aSelectTool` perché erediteranno da `aDevelopProgram` la sola disponibilità di `DECdesign`. Attività eventualmente istanziate prima che l'esecuzione dell'istanza di `SelectTool` sia terminata, restano connesse mediante `AssociateTools/AssociatedTasks` ad entrambe gli strumenti per la progettazione.

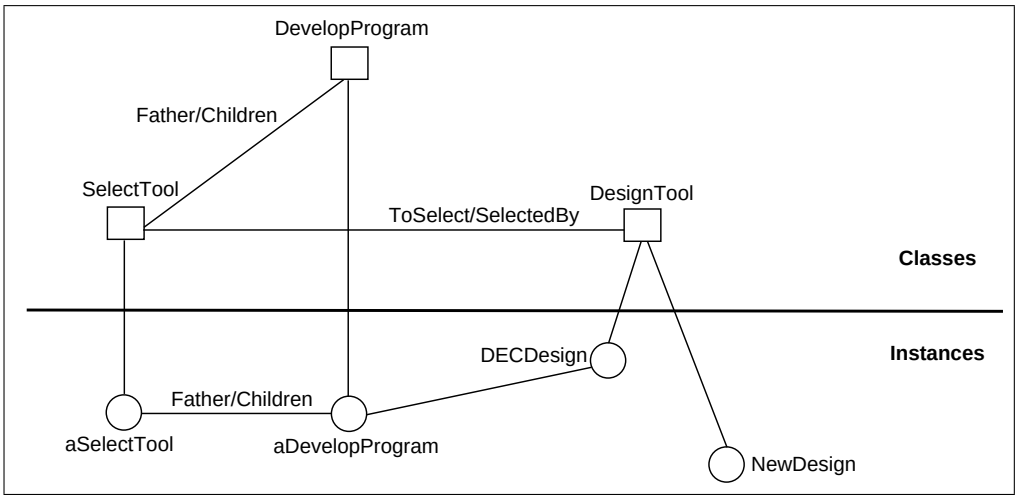

Figura 9.14. `AssociatedTools/AssociatedTasks` dopo l'esecuzione di `SelectTool`

Inserimento di `SelectTool` all'interno di un modello

L'inserimento di `SelectTool` nel modello di un processo richiede che siano tracciate due connessioni indispensabili.

1. All'attributo `Successors` di `SelectTool` deve essere connesso l'attributo `Predecessors` di tutte le sottoclassi di `Task` per cui si vuole sia valida la restrizione imposta dall'esecuzione di `SelectTool`.

2. All'attributo `Father` di `SelectTool` deve essere connesso l'attributo `Children` della classe che decomponendosi genera le attività per cui si vuole imporre la restrizione sull'uso degli strumenti.

È bene avere un meccanismo per specificare per quale tipo di strumento si vuole imporre l'uso di un ben specifico 'tool'. Questo può essere fatto mediante la connessione di livello classe tra l'attributo `ToSelect` della classe `SelectTool` e l'attributo `SelectedBy` della sottoclasse di `Tool` che rappresenta il tipo di strumento per cui si vuole imporre l'uso di uno specifico 'tool'. Se `SelectTool` non si trova connessa a nessuna classe tramite l'attributo `ToSelect`, allora la selezione viene fatta per tutti i tipi di strumento per cui almeno due istanze sono disponibili all'attività che ha istanziato l'oggetto di `SelectTool`. Nella sezione 11.5.5 è riportata una parte del modello di un processo esemplificativo in cui si possono vedere le connessioni cui partecipa la classe `SelectTool`.

10

Aspetti legati alla simulazione in S^3

In questo capitolo è descritto come possono essere realizzati i meccanismi indispensabili per la simulazione dei processi in S^3. In particolare è spiegato come è possibile garantire il corretto ordine di esecuzione tra le attività, il funzionamento dell'istanziazione increamentale, e la gestione del fallimento e riesecuzione delle attività. Inoltre sono proposte alcune ipotesi su come gestire lo scambio di messaggi tra gli oggetti per garantire il funzionamento dei suddetti meccanismi.

Dal momento che le attività sono il fulcro di S^3 e dei suoi modelli di processo, in questo capitolo è descritto nel dettaglio il ciclo di vita di un'attività, mettendo in risalto i tipi di evento che l'attività deve gestire in ognuno degli stati in cui viene a trovarsi.

In questo capitolo è riportata anche una spiegazione di come avviene l'istanziazione dei modelli di processo eseguibili realizzati con la tecnica lagata a S^3.

Infine sono fatte alcune considerazioni sulla gestione dei flussi di esecuzione, non solo per quanto riguarda la realizzazione di S^3, ma anche avanzando ipotesi su come affrontare il problema in un ambiente per l'attuazione dei processi, eventualmente distribuito. Le considerazioni fatte in quest'ultimo ambito permettono di mettere in evidenza come il progetto fatto per S^3 possa facilmente essere utilizzato nella realizzazione di ambienti PM più completi.

10.1 Ordine di esecuzione delle attività

Nella modellizzazione dei processi è indispensabile poter imporre un'ordine tra le varie attività, in modo da poter legare il comportamento di un'attività allo stato di altre attività e quindi allo stato di avanzamento del processo. Tutte le condizioni ed i vincoli che ne nascono devono essere fatti rispettare dal sistema che supporta l'attuazione del modello.

Nel progetto di S^3 si è considerato un solo tipo di precedenza che impone che un'attività non possa cominciare la prorpia esecuzione prima che siano terminate, cioè si trovino nello stato `Completed`, tutte le attività che le sono connesse mediante `Successors/Predecessors`. Tali condizioni sull'esecuzione di un'attività sono espresse mediante un'opportuna semantica associata a determinate connessioni di istanza.

A livello di modello, per imporre la sequenzialità di due attività, è sufficiente tracciare una connessione tra l'attributo `Successors` della prima e quello `Predecessors` della seconda. Sarà quest'ultima a decidere quando può partire la propria esecuzione controllando se tutte le attività connesse mediante il proprio attributo `Predecessors` sono terminate.

Tracciando la connessione tra `Successors` di `T1` e `Predecessors` di `T2` nel modello del processo, si dice solamente che le istanze della classe `T2` devono cominciare la loro esecuzione dopo che quelle della classe `T1` la hanno terminata, ma non si sa di preciso quali siano queste istanze. Per determinarle consideriamo una prima imposizione, cioè che le istanze tra cui valgono le relazioni di ordine siano generate dalla stessa attività, basandosi ancora una volta su di un crirerio di tipo gerarchico nella realizzazione di un meccanismo per la modellizzazione dei processi. Dunque, come è detto nella sezione 10.2, deve essere il padre, nel momento in cui genera gli oggetti che modellizzano le proprie sottoattività, a connetterli tramite la connessione `Successors/Predecessors` in modo opportuno. I dettagli dell'identificazione delle istanze da connetere mediante relazioni di ordine sono spiegati nella sezione 13.4.6 riguardante l'implementazione della classe `Task`.

Volendo disporre di condizioni più precise che vincolano il comportamento delle attività, è sufficiente tracciare connessioni più complicate tra le classi del modello come affermato nella sezione 11.6.1, ed implementare i meccanismi che ne permettono la gestione. Va evidenziato già a questo punto, sebbene si stia trattando la progettazione, che la suddetta implementazione riguarda semplicemente la classe `Task`, cioè una delle classi predefinite, ed i meccanismi

realizzati sono ereditati dalle sottoclassi senza dover essere necessariamente ridefiniti.

10.2 Meccanismo dell'istanziazione incrementale

Le attività possono decomporsi in sottoattività utilizzando però l'istanziazione incrementale, cioè creando le sottoattività solamente nel momento in cui queste sono pronte ad essere eseguite. La connessione `Successors/Predecessors` definisce una condizione sull'esecuzione delle attività, ma per le assunzioni fatte riguardo all'istanziazione incrementale, pone gli stessi vincoli anche sulla creazione degli oggetti che le rappresentano.

Allora quando un'attività si deve decomporre in sottoattività utilizza un'algoritmo che realizza i seguenti passi:

1. interroga la connessione `Father/Children` di livello classe per conoscere le classi dei propri figli;
2. per ogni classe interroga la propria connessione `Father/Children` di livello istanza per controllare se sono già state create istanze;
3. per ogni classe per cui non siano state create istanze, controlla se ha classi connesse come predecessori mediante la connessione `Successors-/Predecessors` di livello classe;
   - se non ci sono classi connesse, crea le istanze;
   - se ci sono classi connesse controlla, mediante la connessione `Father/Children` di livello istanza, che tutti i figli che sono oggetti di tali classi siano terminati;

     se lo sono, istanzia la nuova sottoattività e la collega, mediante la connessione `Successors/Predecessors`, ai figli individuati al passo precedente, che divengono così le attività che devono precedere quella ora istanziata.

Il procedimento utilizzato da questo algoritmo per determinare le istanze delle classi indicate da `Predecessors` che influenzano l'istanziazione, si basa

sull'ipotesi che i predecessori di un'attività debbano essere componenti della stessa attività composita.

L'algoritmo presentato non può essere utilizzato in tutti i casi di decomposizione, ma solo per quelli in cui il numero delle istanze della sottoattività da istanziare sia noto a priori e quindi imposto dal modello del processo. Quando tale numero dipende dall'esecuzione del processo stesso o dal prodotto delle sue attività, si deve utilizzare una variante che è spiegata nel dettaglio nella sezione 13.4.4 che tratta dell'implementazione della classe `Task`.

10.3 La gestione del fallimento delle attività

Non è detto che le attività di un processo giungano sempre ad una corretta terminazione, ma è molto probabile che esse falliscano. Allora il sistema che supporta l'esecuzione deve essare in grado di gestirne il fallimento. Poiché la non corretta terminazione delle attività è principalmente dovuta ai dati che queste ricevono in ingresso, l'azione che più frequentemente segue ad un fallimento è la riesecuzione delle attività che hanno prodotto tali dati.

L'ambiente per la modellizzazione dei processi deve allora mettere a disposizione degli strumenti per modellizzare questo fatto ed anche dei meccanismi per gestire situazione di questo genere.

10.3.1 Connessione `WaitingFeedback/GivingFeedback`

La connessione `WaitingFeedback/GivingFeedback` definita tra sottoclassi di `Task` soddisfa il primo dei due requisiti richiesti per un sistema di modellizzazione dei processi.

Nella figura 10.1 è riportato il modello del semplice processo che regola il ciclo scrittura ('edit'), compilazione ('compile') e costruzione dell'eseguibile ('link') tipico della programmazione che utilizza linguaggi compilati. Questo modello deve tener conto del fatto che se compilazione o collegamento falliscono, il sorgente deve essere corretto.

Poiché il fallimento dell'attività `Compile` richiede la riesecuzione dell'attività `Edit`, si traccia una 'instance connection' tra l'attributo `WaitingFeedback` della classe `Compile` e l'attributo `GivingFeedback` della classe `Edit`.

Questa connessione ha significato sia a livello classe che a livello delle istanze. A livello classe dà un'informazione per quanto riguarda il modello del processo, affermando che il fallimento di un'attività di tipo `Compile`

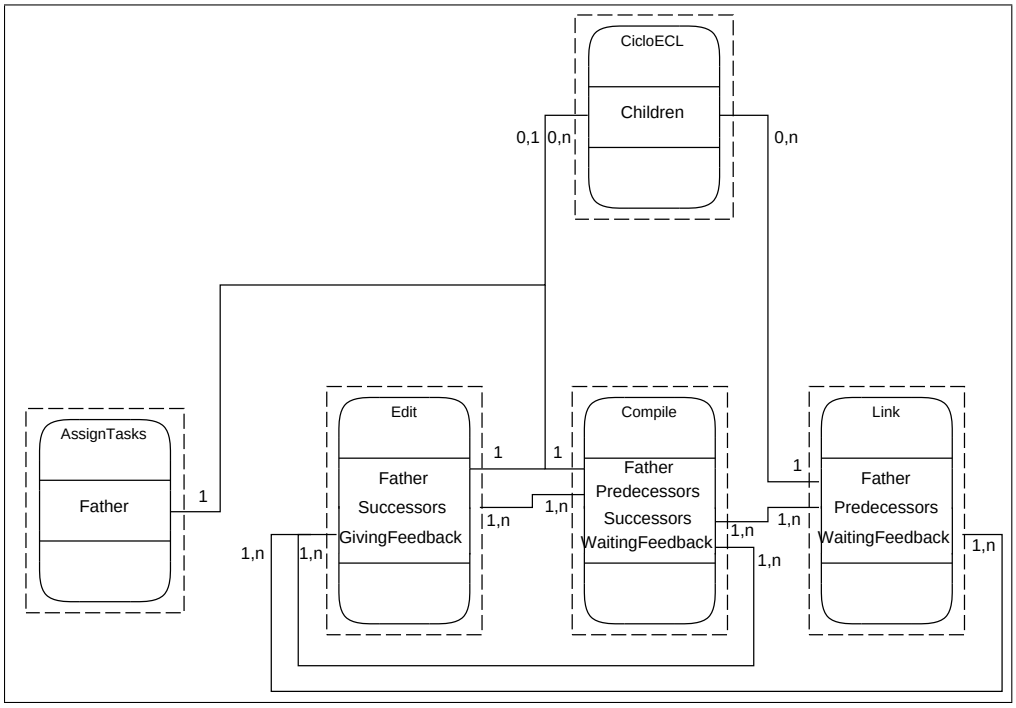

Figura 10.1. Modellizzazione del ciclo Edit-Compile-Link

deve portare alla riesecuzione di una o più attività di tipo `Edit`. Quali istanze di quest'ultima classe debbano effettivamente essere rieseguite, dipende dall'attuazione del processo (in questo caso particolare dalla struttura del prodotto del processo) e sono individuate dal livello delle istanze di questa connessione. In modo analogo a quanto avviene per la connessione `Successors/Predecessors`, quando l'attività padre (`Program`) istanzia le proprie sottoattività le connette opportunamente mediante le connessioni `WaitingFeedback/GivingFeedback` di livello istanza come spiegato nella sezione 13.4.6. Quali siano le istanze da connettere dipende principalmente dal tipo di decomposizione in sottoattività e quindi in casi particolari potrebbe essere una caratteristica peculiare dell'attività composita.

10.3.2 Meccanismi di gestione della riesecuzione

I meccanismi realizzati per gestire la riesecuzione di alcune attività dovuta al fallimento di altre, si basano su alcune ipotesi che si possono verosimilmente

considerare valide in ogni caso utile.

Ipotesi di base

In primo luogo si suppone che, se il fallimento di `T1` causa la riesecuzione di `T2`, allora al momento della riattivazione `T2` si deve trovare in uno stato successivo all'esecuzione, cioè, con riferimento alla figura 9.4, in `Handling-Children` o in `Completed`. Infatti, come detto in precedenza, spesso il problema deriva dai dati prodotti da `T2` che non possono certo essere stati elaborati da `T1` se non erano pronti.

Questo fatto porta ad un'ulteriore ipotesi, cioè che tra `T1` e `T2` ci sia una relazione di precedenza diretta o indiretta. Vale a dire che le due istanze siano direttamente connesse da `Successors/Predecessors` (ad esempio `Edit` e `Compile` nella figura 10.1), oppure che ci sia una catena di queste connesioni che le unisce (ad esempio `Edit` e `Link` nella figura 10.1). In quet'ultimo caso, tutte le attività intermedie devono essere state completate nel momento in cui si ha il fallimento.

Riesecuzione diretta a causa dell'invocazione di `Again`

Sotto le suddette ipotesi, il fallimento di un'attività fa sì che questa invochi il metodo `Again` di tutti gli oggetti di `Task` collegati all'attibuto `Waiting-Feedback` e passi allo stato `WaitingPreconditions`. Il metodo `Again` riporta l'attività interessata nello stato `WaitingPredecessors` in cui avviene il controllo per accertarsi se sia possibile cominciare l'esecuzione. Se ciò è possibile, l'attività ricomincia la propria esecuzione passando nello stato `Executing`.

Nella figura 10.1 è modellizzato il fatto che, quando un'attività di compilazione fallisce, invoca il metodo `Again` di tutte le istanze di `Edit` che le sono connesse. Queste tornano in stato `WaitingPreconditions` e poi, dal momento che non hanno predecessori, transitano immediatamente nello stato `Executing`. Allora l'istanza di `Compile` fallita non può immediatamente essere rieseguita perché i suoi predecessori (oggetti di `Edit`) sono in esecuzione.

Riesecuzione indiretta a causa dell'invio di `Again` ai predecessori

Analoga è la situazione che si verifica a seguito del fallimento di un'attività di tipo `Link`: questa invia il messaggio `Again` alle istanze di `Edit` che le

sono connesse provocandone la riesecuzione. Questo è un caso in cui l'attività fallita e quelle a cui invia il messaggio di riesecuzione, sono connesse da una relazione di precedenza indiretta, cioè attraverso istanze della classe `Compile`. Tali istanze, per le ipotesi fatte, al momento del fallimento devono essere tutte nello stato `Completed`, ma la riesecuzione delle attività di `Edit` fa sì che non siano più vere le loro precondizioni riportandole nello stato `WaitingPreconditions` come descritto più in dettaglio nella sezione 10.5.5. Le istanze di `Compile` in stato `WaitingPreconditions` impongono alle corrispondenti istanze di `Link` di rimanere nello stesso stato così che la riesecuzione dell'attività fallita non avrà luogo fino a che non siano terminate tutte le attività di cui si è forzata la riesecuzione.

Riesecuzione delle sottoattività

Per quanto rigurda le attività che si siano decomposte, la loro riesecuzione implica la riesecuzione di tutti le loro sottoattivtà che deve però avvenire rispettando le relazioni di precedenza tra esse. Questo implica l'invocazione di `Again` non su tutti i figli, ma solamente su quelli che non hanno connessioni di tipo `Predecessors` a livello classe. Infatti la riesecuzione di queste attività, grazie alle notifiche di cambiamento di stato che ad essa seguiranno, causerà anche la riesecuzione, di quelle allo stesso livello nella gerarchia che la seguono. Tale riesecuzione avverrà solamente al completamento delle attività che le precedono, secondo le modalità espresse nella sezione 10.5.2. Un possibile approccio alla realizzazione di questo meccanismo è descritto nella sezione 13.3.6 che tratta dell'implementazione dei metodi della classe `Task`.

Utilizzo di `FeedbackDocument`

Quando un'attività fallisce, genera un rapporto contenente i motivi del fallimento che sono generalemente legati ai dati che riceve in ingresso. Questo rapporto deve essere utilizzato dallle istanze di `Task` che vengono fatte ripartire a causa del fallimento dell'attività che lo ha generato.

Il modello del processo deve prevedere il fatto che le attività che possono essere rieseguite, ricevano in ingresso il rapporto che documenta le cause del fallimento degli oggetti `Task` connessi mediante `GivingFeedback`. In termini della tecnica di modellizzazione mostrata in questa trattazione, ciò significa che deve esistere una connessione `Inputs/Users` tra la classe di queste attività e `FeedbackDocument`. Quando le istanze di queste sottoclassi di `Task`

sono eseguite la prima volta, non esiste nessun oggetto `FeedbackDocument` che le interessa, cioè che sia connesso mediante `ReferredTo/FaultsDocument` con qualcuno dei dati che esse hanno in uscita o in ingresso. Quando invece si ha una riesecuzione dovuta al fallimento di una delle attività connesse mediante `GivingFeedback`, è stata creata un'istanza di `FeedbackDocument` che risulta connesa ad uno degli oggetti `Data` in ingresso o in uscita, e quindi anche questa viene collegata mediante la connessione `Inputs/Users` di livello istanza. Durante l'esecuzione dell'attività in questione su tutti gli oggetti individuati da `Inputs` viene lanciato uno strumento per utilizzarli, e lo stesso avviene con l'istanza di `FeedbackDocument`.

10.4 Lo scambio di messaggi

Nella sezione 10.1 si è spiegato come le condizioni che vincolano l'esecuzione delle attività siano espresse nel modello mediante il tracciamento di `instance connection` tra le classi corrispondenti. Questo fatto non stabilisce come gli oggetti interessati siano in grado di determinare se le loro precondizioni siano soddisfatte o meno; si può affrontare la questione in alcuni modi differenti.

1. Quando un'attività cambia il proprio stato lo comunica a tutte le attività connesse mediante gli attributi `Successors` e `Father` invocandone il metodo `HandleTransition` che provvede a realizzare gli opportuni controlli ed eventualmente a forzare un cambiamento di stato. Per decidere se sia necessario effettuare un cambiamento di stato è necessario conoscere lo stato di tutte le attività connesse da `Predecessors` e `Children`, e non solo di quella che ha segnalato la propria transizione. Si può agire in due modi differenti:

   (a) il metodo `HandleTransition` memorizza all'interno dell'oggetto lo stato del mittente (se ha finito la sua esecuzione, oppure no) e ad ogni chiamata controlla se tutti i `Predecessors` o `Children` sono in stato `Completed`;

   (b) il metodo `HandleTransition` interroga tutti gli oggetti connessi tramite `Predecessors` o `Children` per controllarne lo stato. Quindi in base alle risposte effettua o meno la transizione.

2. Ogni attività è modellizzata da un oggetto attivo, cioè dotato di un suo flusso di esecuzione, che periodicamente richiede a tutte le istanze di `Task` che gli sono connesse mediante gli attributi `Predecesors` e `Children`, il loro stato. In base alle risposte ottenute mediante le invocazioni di metodo corrispondenti, stabilisce se rimanere nello stato in cui si trova, o se effettuare una transizione.

3. Ogni istanza di `Task` comunica solamente con il proprio padre informandolo di ogni suo cambiamento di stato. Questa informazione va fornita in ogni caso perché lo stato del padre dipende anche dallo stato delle sue sottoattività. È il padre a controllare quando ciascuno dei suoi figli deve cambiare stato e, se è il caso, comunica che deve avvenire la transizione.

La seconda soluzione è sconsigliata perché non si adatta bene né a sistemi con un solo flusso di controllo, né ad ambienti ad alto livello di distribuzione; infatti l'intenso traffico di messaggi può rivelarsi una insormontabile limitazione alla velocità del sistema.

Lo svantaggio della terza soluzione sta nel fatto che il padre deve conoscere e controllare le precondizioni dei figli e questo non è coerente con l'incapsulamento, uno dei principali punti di forza della metodologia ad oggetti. Infatti dovendo cambiare le precondizioni di un'attività non si deve modificare un'informazione all'interno dell'attività stessa, ma all'interno del padre.

Per questi motivi nella realizzazione di S^3 si è scelta la prima soluzione che si adatta bene ad un ambiente mono processore come quello del prototipo presentato nella parte IV. Risulta comunque immediata l'espansione ad un ambiente distribuito come accennato nella sezione 10.7.3. La prima soluzione permette due varianti, la prima delle quali implica tenere memorizzato in un oggetto lo stato di altri. Questo causa due inconvenienti che sono uno la conseguenza dell'altro:

1. si ha duplicazione di informazione;

2. si rischia di avere, a causa di malfunzionamenti, incoerenza tra le informazioni nei due siti.

Nonostante ciò la variante utilizzata nell'implementazione di S^3 è proprio questa. Essa è di più immediata realizzazione e facilmente convertibile nell'altra variante; l'informazione duplicata è poca e quindi poco ingombrante

ed infine è difficile, in un ambiente mono processore come quello del prototipo di sistema per la simulazione dei processi realizzato, avere anomalie di funzionamento che portino a stati di incoerenza.

10.5 Il ciclo di vita di un'attività

Nella sezione 9.1.3 si è descritto brevemente il diagramma degli stati degli oggetti della classe `Task` che è stato riportato per comodità nella figura 10.2; esso descrive in quale modo evolva la vita di un'attività durante l'attuazione del processo e quali eventi provocano la transizione da uno stato all'altro.

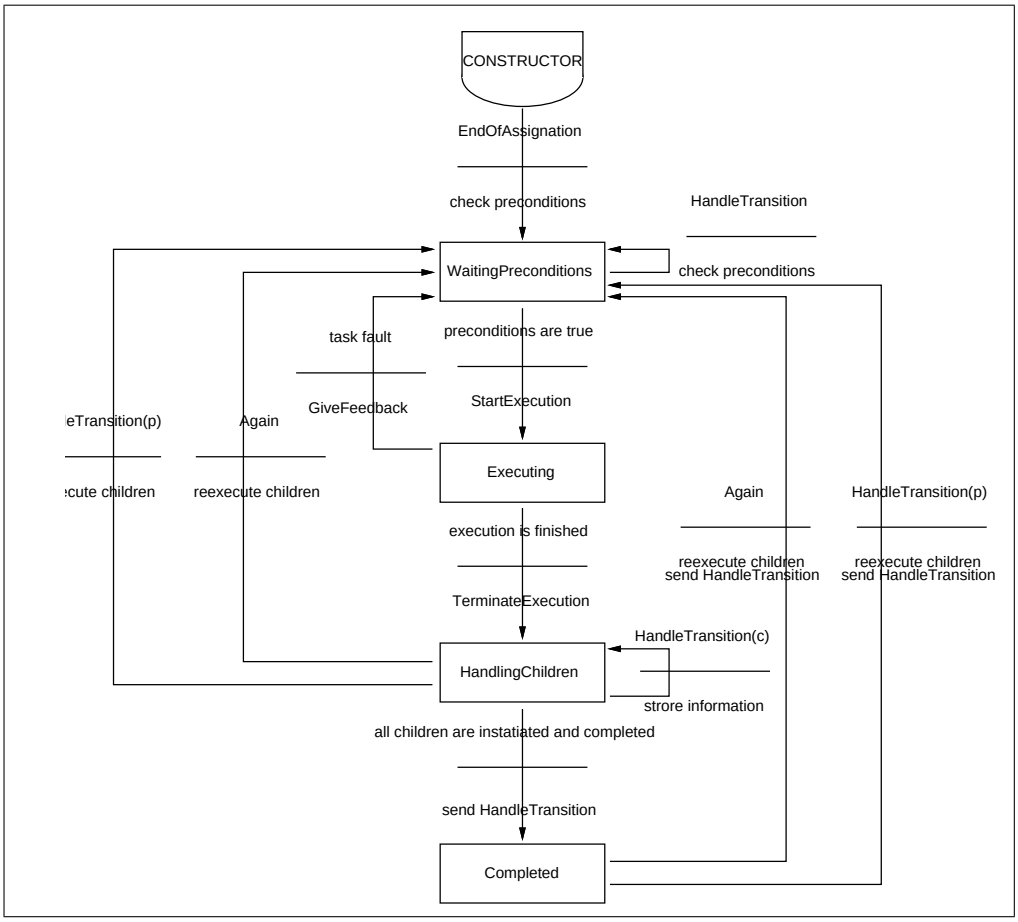

Figura 10.2. Diagramma degli stati degli oggetti di `Task`

10.5.1 Creazione dell'attività

Quando un'istanza di `Task` è stata creata essa permane nello stato `Constructor` fino a che non è pronta ad essere eseguita. Infatti, come accennato nella sezione 9.1.3, un'attività, pur essendo terminate quelle che la precedono, non può essere eseguita fino a che non le sia stato assegnato un responsabile. Questo si basa sul presupposto che qualsiasi attività, anche quando sembri essere automatica, necessiti di interagire con una persona, soprattutto per la gestione di anomalie o malfunzionamenti.

Allora, dopo avere istanziato le sottoattività, il padre si occupa, mediante i meccanismi descritti nella sezione 9.2.2, di assegnare loro i rispettivi responsabili e le eventuali persone messe a disposizione. A questo punto viene invocato il metodo `EndOfAssignation` che informa il ricevente che le operazioni suddette sono state portate a termine e che l'attività può cominciare la propria esecuzione se le condizioni che la vincolano sono soddisfatte.

Precondizioni che vincolano l'istanziazione e l'esecuzione

Questo è espresso nel digramma di stato dalla transizione verso lo stato `WaitingPreconditions` che ha come evento `EndOfAssignation` e come azione il controllo delle precondizioni. L'oggetto permane in questo stato fino a che le precondizioni per la sua esecuzione non sono soddisfatte. In casi standard queste precondizioni sono il completamento delle attività connesse mediante l'attributo `Predecessors`. Però, dal momento che è l'oggetto stesso che le controlla, si possono aggiungere altri tipi di condizioni necessarie in casi particolari.

Il meccanismo dell'istanziazione incrementale prevede che l'oggetto che modellizza un'attività non sia creato fino a che questa non sia in grado di essere eseguita. Questa abilitazione all'esecuzione è rappresentata dalla soddisfazione delle condizioni standard, cioè dal fatto che tutti i predecessori abbiano terminato la loro esecuzione. In ogni caso è necessario che l'oggetto appena creato passi dallo stato `Constructor` allo stato `WaitingPreconditions` e non direttamente ad `Executing` per due motivi.

1. Durante l'assegnazione delle persone all'attività, che avviene dopo che l'istanza di `Task` è stata creata, le condizioni che prima erano soddisfatte, possono divenire non più vere e quindi non è detto che l'esecuzione possa avere luogo. Ad esempio un'attività collegata mediante l'attributo `Predecessors` che al momento dell'istanziazione era terminata, può

essere stata rieseguita a causa di una notifica di fallimento, e quindi non esserte più in stato `Completed`.

2. Può essere necessario avere condizioni per l'esecuzione più complesse di quelle controllate dal padre nel fare l'istanziazione incrementale. Allora tali condizioni possono essere controllate dall'oggetto interessato durante la permanenza nello stato `WaitingPreceonditions` senza che debba necessariamente essere modificata l'implementazione del padre.

Se le precondizioni aggiuntive possono richiedere molto tempo prima che siano soddisfatte, l'istanziazione incrementale ne deve tenere conto perché diversamente diventerebbe poco efficiente. Questo richiede di modificare il codice che regola l'istanziazione nell'oggetto padre per poter variare le condizioni che vincolano la creazione del figlio. Ciò non è completamente in linea con il principio dell'incapsulamento proprio dell'approccio ad oggetti.

Anche il fatto che dopo la sua creazione, un'attività non possa essere immediatamente eseguita perché devono prima essere fatte le assegnazioni delle persone, può rendere inutile l'uso dell'istanziazione incrementale. In realtà è piuttosto improbabile che uno dei predecessori sia rieseguito nel tempo in cui vengono assegnate le persone. Infatti l'assegnazione è un'attività di durata breve rispetto alle altre attività tipiche dei processi per la produzione del software che poterbbero fallire e causare la riesecuzione. È evidente che il meccanismo per l'istanziazione incrementale dovrebbe tenere conto anche dell'assegnazione delle persone, ma questo complicherebbe notevolmente la decomposizione in sottoattività e soprattutto è parso non rientrare tra gli scopi di questa trattazione.

10.5.2 Verifica delle precondizioni

Le attività che passano nello stato `Completed` o che da questo escono, invocano il metodo `HandleTransition` del padre e degli oggetti `Task` connessi mediante l'attributo `Successors`. Secondo il meccanismo descritto nella sezione 10.4 l'oggetto che riceve il messaggio memorizza il nuovo stato del mittente e quindi intraprende le azioni opportune.

- Se l'attività terminata è connessa mediante l'attributo `Children` è necessario controllare se sia possibile istanziare nuove sottoattività a seguito dell'avvenuta terminazione.

- Se l'attività terminata è connessa mediante l'attributo `Predecessors`, l'oggetto che ha ricevuto il messaggio deve controllare se debba rimanere nello stato in cui si trova o eseguire una transizione.

In quest'ultima situazione, se il ricevente si trova nello stato `WaitingPreconditions`, controlla se tutti gli oggetti connessi mediante `Predecessors` sono nello stato `Completed`. Se questo si verifica si ha il passagio allo stato `Executing` invocando il metodo `StartExecution`.

Il Comportamento appena descritto è visibile nel diagramma degli stati riportato nella figura 10.2 in cui si ha una transizione che ha origine e destinazione nello stato `WaitingPreconditions` che è scatenata dall'evento `HandleTransition(p)`. Questa notazione sta ad indicare che l'attività che ha notificato il proprio completamento è una delle istanze connesse mediante `Predecessors`; l'unico effetto di questo evento, dal momento che non provoca direttamente un cambiamento di stato, è l'attivazione dei controlli per verificare che le precondizioni di esecuzione siano soddisfatte. Se tali precondizioni sono vere, si ha l'innescamento della transizione che porta allo stato `Executing` provocando come azione l'invocazione del metodo `StartExecution`. Si può facilmente notare che questa transizione non è etichettata da un evento, ma dalla condizione che la attiva; infatti il passaggio di stato che descrive non è scatenato da un messaggio proveninte da un altro oggetto, ma dall'esito dei controlli fatti da parte dell'oggetto stesso.

Può accadere che, mentre l'attività in esame si trova in questo stato, venga invocato il suo metodo `Again`; tale evento viene ignorato perché non appena possibile comincierà l'esecuzione. Questa è comunque una circostanza molto particolare perché può derivare solo da un fallimento di un'altra attività che sia avvenuto dopo che una precedente invocazione di `Again` abbia già riportato l'attività in esame nello stato `WaitingPreconditions`.

10.5.3 Esecuzione

Il metodo `StartExecution` contiene il codice che realizza le funzioni iniziali della specifica attività. In primo luogo esso deve istanziare eventuali sottoattività, se il modello lo prevede e se il processo presenta le condizioni necessarie affinché ciò possa avvenire.

Le azioni che costituiscono l'esecuzione di un'attività sono di uno dei tipi elencati nel seguito; nella figura 10.2 non si sono riportati gli eventi e le azioni

ad essi conseguenti, che fanno comunque restare nello stato `Executing`, per non complicare eccessivamente il diagramma.

Attesa della disponiblità dell'utente ad interagire

Ogni volta che un'attivtà ha necessità di interagire con l'utente, invoca il metodo `AskForUserAttention` dell'oggetto `Person` che modellizza il proprio responsabile. A questo punto il mittente attende che la persona interessata comunichi la propria disponibilità invocando il metodo `GotUserAttention` dell'oggetto `Task`.

Perché questo meccanismo possa funzionare è necessario che il sistema preveda un'interfaccia di dialogo con l'utente mediante cui egli possa essere notificato delle attività che richiedono il suo intervento e possa, quando lo ritiene necessario, comunicare la propria disponibilità. Sarà proprio questa interfaccia a comunicare all'oggetto `Person` che modellizza la persona interessata, la disponibilità da parte dell'utente; di tale disponibilità sarà quindi notificata l'attività invocandone il metodo `GotUserAttention`.

Nella progettazione ad oggetti di S^3 si è utilizzata la metodologia ad oggetti Coad/Yourdon, ma del modello si presenta solamente il componente del dominio del problema ('Problem Domain Component'). Poiché l'interfaccia di cui si è appena parlato fa invece parte del componente dell'interazione con gli umani ('Human Interaction Comonent') non se ne parla dettagliatamente a questo punto, ma è descritta con maggiore dettaglio nella sezione 13.1.2 trattando della sua implementazione.

Esecuzione di interfacce per l'interazione con l'utente

Per interagire con l'utente il processo si avvale di apposite interfacce che permettono al responsabile di comunicare le proprie decisioni all'attività che, in base a queste ultime, può intraprendere le azioni opportune.

Azioni tipiche che richiedono l'interazione con una persona sono la scelta di uno o più elementi all'interno di una lista (ad esempio l'assegnazione delle persone o la scelta degli strumenti automatici da utilizzare), oppure decisioni riguardo la bontà o meno di dati prodotti da attività precedenti (ad esempio la revisione del progetto). Per ognuna di queste azioni viene generata ed eseguita sulla macchina dell'utente interessato, un'interfaccia che permetta alla persona di esprimere ciò che ritiene più adeguato. L'interfaccia stessa si fa carico di comunicare le decisioni dell'utente all'attività che la ha eseguita;

i dettagli di questo meccanismo non sono trattati a questo punto perché non rientrano nelle compenze del componente del dominio del problema.

Esecuzione di strumenti automatici

Per generare i propri prodotti le attività possono avvalersi di strumenti automatici ('tool') che sono in grado di svolgere il loro compito in modo del tutto autonomo, oppure interagendo con una persona che sarà, come sempre, il responsabile dell'attività in questione.

Quindi in un ambiente per l'attuazione dei processi software, gli oggetti `Task` devono essere in grado di:

1. determinare gli strumenti da eseguire per generare i dati che vanno prodotti,

2. lanciarne l'esecuzione sulla macchina opportuna.

Nella modellizzazione di processi in S^3 si utilizzano le sottoclassi di `Tool` per rappresentare gli strumenti automatici; nella classe `Tool` è definito il metodo `Run` che si occupa dell'esecuzione del programma corrispondente. A livello di modello si trascura il secondo dei due punti mostrati sopra, perché, grazie all'incapsulamento offerto dall'orientamento agli oggetti, esso diviene responsabilità degli oggetti di `Tool` e deve essere affrontato nell'implementazione della classe. In realtà, dal momento che S^3, è un ambiente per la simulazione dei processi software, è inutile che avvenga l'esecuzione degli strumenti automatici.

Gestione della terminazione di una sottoattività o di un predecessore

Le azioni che devono essere intraprese ogni qualvolta un'attività comunica un cambiamento di stato sono le stesse descritte per lo stato `WaitingPreconditions`. In base alle assunzioni fatte nella sezione 10.1 sulla gestione delle condizioni di esecuzione delle attività, la notifica di un cambiamento di stato tramite il messaggio `HandleTransition`, viene mandata da un'attività solamente quando entra nello stato `Completed` o quando da questo stato esce. Queste ipotesi semplificano notevolmente la gestione delle notifiche di cambiamento di stato che è realizzata secondo quanto descritto nella sezione 13.3.3 trattando l'implementazione della classe `Task`.

Il metodo `HandleTransition` deve in primo luogo memorizzare l'avvenuto cambiamento di stato per poterne tenerne conto in successivi controlli delle condizioni. Quindi deve agire in modo differente a seconda che chi ha invocato il metodo sia connesso mediante l'attributo `Children` o mediante l'attributo `Predecessors`. Nel primo caso deve controllare se la terminazione dell'attività in questione permette di istanziare nuovi figli.

Il secondo caso deriva da una situazione particolare in cui è venuto a trovarsi il processo. Infatti l'invocazione del metodo `HandleTransition` da parte di un'attività `T1` predecessore di `T2` mentre `T2` è in esecuzione, implica che `T1` sia stato rieseguito a causa del fallimento di un'attività `T3` che è connessa a `T1` mediante `WaitingFeedback/GivingFeedback`. Questo significa che `T2` e `T3` sono entrambe successori di `T1`, ma non hanno una relazione d'ordine che le lega e quindi sono andate in esecuzione contemporaneamente.

Le azioni che devono essere intraprese da `T2` dipendono dalla politica che si vuole utilizzare nella gestione di questa particolare situazione. Infatti probabilmente, dal momento che `T2` è successore di `T1`, ne utilizza i prodotti; quindi la riesecuzione di `T1` lascia supporre che quest'ultima modificherà i dati che produce e quindi che `T2` stia operando su dati che saranno presto obsoleti. Si possono scegliere diversi modi di affrontare questa situazione.

- Si lascia continuare l'esecuzione di `T2` che comunque, una volta terminata, dovrà essere ricominciata.
- `T2` notifica l'utente del fatto e quindi può:
  - continuare l'esecuzione;
  - permettere al proprio responsabile di scegliere se continuare l'esecuzione oppure interromperla;
  - interrompere l'esecuzione.

La prima soluzione è la più semplice ed immediata ed è quella che viene adottata in S^3. Sicuramente non è la più funzionale, ma la gestione di questa situazione è una questione indipendente dal modello del processo e che non rientra tra gli obiettivi di questa trattazione.

Analoga situazione si ha quando a `T2` giunga una nuova invocazione di `HandleTransition` da parte di `T1`, indicando questa volta che l'esecuzione di `T1` è terminata. Questo significa che la nuova versione dei dati prodotti da `T1` è pronta e quindi, alle possibili azioni che `T2` può intraprendere elencate in

precedenza, si aggiunge la possibilità di continuare l'esecuzione sulla nuova versione dei dati di ingresso. In S^3 l'esecuzione di `T2` è lasciata continuare, per cui l'invocazione di `HandleTransition` deve memorizzare nell'oggetto ricevente il fatto che esso debba essere rieseguito. Infatti a questo punto `T1` risulta nuovamente terminato come all'inizio dell'esecuzione di `T2` ed in nessun modo quest'ultimo potrebbe sapere che i dati da lui prodotti non sono più validi. A tutte queste questioni è legato il problema della gestione delle versioni dei dati che è affrontato più nel dettaglio nella sezione 8.9.

Riesecuzione imposta dal padre

Come detto in precedenza, il fallimento di un'attività causa la riesecuzione di tutte quelle connesse ad essa mediante `WaitingFeedback` sulle quali viene invocato il metodo `Again`. poiché la riesecuzione di un'attività, come detto nella sezione 10.5.3, implica la riesecuzione di tutte le sottoattività, in primo luogo viene invocato il metodo `Again` dei figli secondo le modalità espresse nella sezione 10.5.3 in modo da rispettare le relazioni d'ordine tra le sottoattività.

Si tratta a questo punto di decidere come gestire la riesecuzione di un'attività che sia in stato `Executing` avendo a disposizione alternative simili a quelle presentate per l'analogo problema, dovuto alla riesecuzione di un predecessore, discusso poco sopra. Anche in questo caso si può tener presente che si tratta di decisioni che sono indipendenti dal modello di processo ed al di fuori degli intenti di questa trattazione.

Vale la pena di evidenziare, rispetto al caso precedente, che questa situazione si verifica quando la riesecuzione è richiesta esplicitamente a seguito del fallimento di un'attività che deve causare la riesecuzione di quella in questione; questo è imposto dal modello perché le due sono connesse da `WaitingFeedback/GivingFeedback`, per cui è bene, qualsiasi sia la politica adottata, che il responsabile sia avvisato.

In S^3 l'attività su cui è invocato `Again` memorizza l'evento in modo che, uscendo dallo stato `Executing`, ne possa tenere conto tornando nello stato `WaitingPreconditions`.

10.5.4 Gestione delle sottoattività successiva all'esecuzione

Quando l'attività ha eseguito tutte le azioni che ne caratterizzano l'esecuzione, può uscire dallo stato `Executing` invocando su se stessa il metodo `TerminateExection`. Nel caso particolare di attività composite che semplicemente si decompongono in sottoattività, `TerminateExecution` è invocato direttamente da `StartExecution` subito dopo l'istanziazione dei primi figli.

`TerminateExecution` porta l'istanza di `Task` nello stato `HandlingChildren` (come si vede nel diagramma degli stati della figura 10.2) in cui l'esecuzione è terminata, ma l'attività non si può considerare completata, perché non tutte le sue sottoattività sono nello stato `Completed`. Appena si trova nello stato `HandlingChildren`, l'attività deve operare dei controlli (come espresso dall'azione associata alla transizione da `Executing` nel diagramma della figura 10.2) per decidere se può rimanervi o se è necessario operare un cambiamento di stato motivato da tre possibili cause.

1. Uno degli oggetti connessi mediante `Predecessors` non si trova più nello stato `Completed`. Di questo l'attività in questione era stata informata quando si trovava nello stato `Executing`, ma per la politica di gestione adottata nella progettazione di S^3, non si era fatto nulla, se non memorizzare il nuovo stato del predecessore. Allora, a questo punto, l'attività deve tornare nello stato `WaitingPreconditions` perché deve essere rieseguita non appena le precondizioni saranno soddisfatte.

2. Si è memorizzato, come stabilito nella sezione precedente, il fatto che l'attività deve essere rieseguita a seguito della riesecuzione e successivo completamento di uno dei suoi predecessori o a seguito della ricezione di un mesaggio di `Again`. Allora l'attività in questione viene fatta transitare nello stato `WaitingPreconditions` da cui appena possibile, magari anche immediatamente, passerà nello stato `Executing`.

3. Tutti i figli sono stati istanziati e sono in stato `Completed`: l'attività può transitare nello stato `Completed`.

Se invece l'attività resta nello stato `HandlingChildren` essa non fa altro che eseguire i metodi che le sono invocati e cioè `Again` e `HandleTransition`. Quest'ultimo in primo luogo deve determinare se il chiamante è un'oggetto connesso mediante `Children` o mediante `Predecessors`; poiché gli effetti

dell'invocazione sono molto differenti nei due casi, nel diagramma degli stati riportato nella figura 10.2, se il mittente è una sottoattività l'invocazione del metodo è indicata come `HandleTransition(c)`, se è un predecessore si utilizza `HandleTransition(p)`.

Invocazione di `HandleTransition` da parte di sottoattività Indica il completamento o la riesecuzione di una delle sottoattività lo stato della quale è memorizzato e quindi viene aggiornata l'informazione; questo è espresso dalla transizione che ha origine e dedstinazione nello stato `HandlingChildren` attivata dall'evento `HandleTransition(c)`. L'azione associata è appunto l'aggiornamento dei dati memorizzati riguardanti la sottoattività in questione ed il controllo delle condizioni per cambiare stato. Se a questo punto tutti gli oggetti connessi mediante `Children` risultano essere nello stato `Completed`, l'attività che ha ricevuto l'invocazione passa anch'essa nello stato `Completed` come indicato dalla transizione verso questo stato che non è innescata da un evento, ma dalla condizione appena menzionata.

Invocazione di `HandleTransition` da parte di predecessori Indica che una delle attività connesse mediante `Predecessors` non è più nello stato `Completed`; infatti il fatto stesso di essere nello stato `HandlingChildren`, implica che tutti i predecessori debbano essere terminati ed un'invocazione di `HandleTransition` non può che informare della riesecuzione di uno di questi. In questa situazione non si ha alcuna alternativa che riportare l'attività nello stato `WaitingPreconditions`, come viene indicato dalla transizione innescata dall'evento `HandleTransition(p)` nel diagramma degli stati della figura 10.2, curandosi di rieseguire le sottoattività che siano eventualmente già terminate. Dopo che è passata nello stato `WaitingPreconditions`, l'attività sarà rieseguita non appena possibile.

Invocazione di `Again` Può essere fatta dal padre dell'attività in questione perché esso stesso è stato soggetto ad un'invocazione di questo messaggio, oppure da uno degli oggetti `Task` connessi mediante `GivingFeedback` la cui esecuzione sia fallita. Analogamente a quando ci si trova in qualsiasi altro stato, si deve invocare il metodo `Again` per le sottoattività secondo le modalità descritte in breve nella sezione 10.5.3. Dopo di ciò l'oggetto deve transitare nello stato `WaitingPreconditions` per riiniziare, appena le precondizioni siano verificate, l'esecuzione.

10.5.5 Gli eventi gestiti dopo la terminazione

In questo stato l'attività può ricevere gli stessi messaggi che può ricevere quando si trova in `HandlingChildren`, ma l'effetto di alcuni è diverso.

Invocazione di `Again` L'esecuzione del metodo `Again` è la stessa di quella descritta per lo stato precedente; quindi provoca l'invocazione del servizio `Again` per le sottoattività e la transizione nello stato `WaitingPreconditions`. Come evidenziato dall'azione associata alla transizione in questione nel diagramma degli stati riportato nella figura 10.2, dal momento che l'oggetto lascia lo stato `Completed` deve informare il padre ed i successori inviando ad ognuno il messaggio `HandleTransition`.

Invocazione di `HandleTransition` Può provenire solamente da un oggetto connesso mediante `Predecessors` perché le sottoattività sono tutte terminate e non è possibile che una di esse sia riattivata da parte di un'attività che non sia il proprio padre. Infatti il metodo `Again` viene invocato solamente dal padre o da un'istanza connessa mediante `GivingFeedback`, ma si è supposto che tali istanze siano sottoattività di una stessa attività composita.

Poiché l'invocazione proviene da uno dei predecessori che informa di non trovarsi più nello stato `Completed`, l'attività in questione è riportata nello stato `WaitingPreconditions` per essere rieseguita non appena sia possibile. Come nel caso della ricezione di `Again`, è necessario invocare il metodo `Again` sulle sottoattività ed informare padre e successori mediante il messaggio `HandleTansition`.

10.6 L'istanziazione del processo

L'istanziazione del processo software costituisce, unitamente all'attuazione, il passo del meta-processo che segue l'implementazione. Nonostante questo è bene darne un accenno a questo punto della trattazione per chiarirne alcuni aspetti che prescindono dall'implementazione e la influenzano in alcuni particolari.

L'esecuzione (simulata) del processo comincia con la creazione di un'istanza della classe che rappresenta l'attività di più alto livello, e che quindi

rappresenta l'intero processo. Questa attività comincia quindi la propria esecuzione decomponendosi nelle sottoattività che la costituiscono; perché ciò sia possibile, non è suffuciente creare l'oggetto che rappresenta il processo, ma anche connettere ad esso le risorse di cui ha bisogno per portare a termine la propria esecuzione.

Per poter cominciare l'esecuzione del processo è allora necessario creare gli oggetti che modellizzano le persone coinvolte, gli strumenti automatici che vengono messi a disposizione delle varie attività per operare sui dati ed i requisiti del prodotto da realizzare. Poiché i primi due tipi di oggetti possono essere condivisi da più processi, non è detto che debbano essere creati ogni volta che si istanzia un nuovo processo, ma in un ambiente per la modellizzazione ed attuazione dei processi, possono essere recuperati da una base di dati che ne garantisce la persistenza oltre la vita dei singoli processi.

Un ambiente di questo genere metterà anche a disposizione del 'project manager', incaricato dell'istanziazione del processo, un'opportuna interfaccia grafica che gli permetta di scegliere tra le risorse attualmente a disposizione del sistema, o di crearne di nuove, per assegnarle al processo che si sta istanziando.

Nell'associare le persone alla nuova istanza dell'attività di più alto livello, è necessario specificare il ruolo che queste ricoprono mediante le connessioni con oggetti `Role` descritte nella sezione 9.2.1. Anche per questo scopo è bene avere un'interfaccia per l'istanziazione che dia supporto al responsabile del progetto.

10.7 I flussi di esecuzione

Tutto ciò che è stato presentato circa il progetto di S^3, non pone nessuna condizione sulla gestione dei flussi di esecuzione ('*control thread*'). Il progetto realizzato per S^3 si adatta sia ad un'esecuzione che utilizza un singolo flusso di controllo (mono-processo) sia ad una che utilizzi la concorrenza, sia questa su un unico processore o in un ambito distribuito.

Nelle sezioni successive sono prese in considerazione varie ipotesi su come trattare i flussi di esecuzione nell'implementazione del progetto presentato. Le considerazioni che saranno fatte non riguardano strettamente la realizzazione di un ambiente per la simulazione dei processi come S^3, ma eventualmente anche uno per la loro attuazzione. Infatti, il progetto presentato in questa trattazione può essere utilizzato efficacemente per passare alla

realizzazione di un vero e proprio ambiente PM.

Trattando dei flussi di esecuzione si parla spesso di *processo* nel senso più strettamente informatico del termine, cioè stando ad indicare l'esecuzione di un certo numero di istruzioni che costituiscono un programma. Bisogna quindi fare attenzione a non confondere il processo in quanto programma in esecuzione, con il processo software.

10.7.1 Singolo processo

Una prima possibile implementazione consiste nel realizzare l'attuazione del processo software su di un unico processore e con un approccio sequenziale. Si ha cioè un solo 'control thread' che passa da un oggetto all'altro quando ne vengono invocati i metodi; si tratta esattamente dello stesso meccanismo utilizzato da un linguaggio di programmazione sequenziale in cui si abbiano chiamate a procedura.

Questa soluzione, sebbene sia la più semplice ed immediata, non è molto adatta per l'attuazione di processi software che è intrinsecamente distribuito e si presta bene a soluzioni distribuite che aumentano certamente le prestazioni dell'esecuzione.

Per realizzare l'ambiente progettato e le descrizioni eseguibili dei processi secondo la soluzione mono-processo, è sufficiente avere a disposizione un qualsiasi linguaggio di programmazione sequenziale. Non è necessario che questo sia ad oggetti, infatti l'analisi e la progettazione ad oggetti non impongono un'implementazione ad oggetti. È comunque evidente che una implementazione ad oggetti sarebbe più facilmente realizzabile in modo efficiente e conforme al modello ottenuto.

10.7.2 Processi indipendenti per l'interazione con l'utente

Una soluzione analoga alla precedente un processo principale che coordina l'attuazione del processo software, ma ogni interfaccia necessaria per la comunicazione con l'utente ed ogni strumento automatico che viene lanciato, dispongono di un loro proprio flusso di esecuzione. Poiché si sta pur sempre parlando di utilizzare un singolo processore, tra i vari processi non si avrà concorrenza vera e propria, ma simulata mediante la suddivisione del tempo di utilizzo del processore da parte dei vari processi attivi ('time slicing').

Estensione con uso della distribuzione

La soluzione che prevede processi concorrenti per l'interazione con l'utente, rispetto a quella mono-processo, non presenta particolari vantaggiin termini di prestazioni fino a che è utilizzata con un singolo processore, ma si può adattare in modo praticamente immediato ad architetture che prevedano più processori o addirittura più macchine collegate tra loro. L'unica differenza sarà nei meccanismi di comunicazione tra i vari processi i quali si troveranno a questo punto ad esere eseguiti da processori differenti.

Versione iniziale di S^3

Seguendo questo approccio che prevede processi concorrenti per l'interazione con l'utente nella variante a singolo processore, è stata realizzata la prima versione di S^3. Esso però è un abiente per la simulazione, e non l'attuazione, dei processi software e ciò implica, tra l'altro, che non ci si preoccupi della produzione vera e propria dei dati da parte delle attività. Questo fatto permette di non si gestire l'archiviazione dei dati e l'esecuzione degli strumenti automatici che è solamente simulata.

Dal momento che S^3 è stato realizzato mediante il linguaggio ad oggetti Smalltalk-80, si è sfruttata la gestione dei procesi concorrenti che questo mette a disposizione ed ogni qualvolta si rende necessario eseguire un'interfaccia, la si affida ad un nuovo processo. Quando questa ha finito la propria esecuzione, il processo corrispondente viene ucciso. Non si ha cominicazione tra processi mediante scambio di messaggi, ma essi scrivono in zone di memoria comune, ad esempio lo stesso attributo di un oggetto, che sono protette da semafori utilizzati per ottenere mutua escusione.

Possibile implementazione in ambiente distribuito

Volendo realizzare in un ambiente distribuito un approccio che prevede processi concorrenti per la gestione dell'interazione con le persone, viene immediata la soluzione che vede il processo principale eseguito su una macchina, magari dedicata; con questa macchina comunicano quelle sulle quali sono eseguite le interfacce per l'interazione con l'utente e gli strumenti automatici. Bisogna tener presente che il processo centrale non ha bisogno di un calcolatore particolarmente potente per la sua esecuzione, perché le attività

dei processi software hanno tutte durate molto grandi e gli eventi che caratterizzano l'attuazione del processo non si susseguono ad elevata frequenza. Questo implica che tale processo principale si troverebbe molto spesso in stato di attesa di eventi (messaggi provenienti dalle altre macchine) e solo all'arrivo di questi passerebbe ad una fase di esecuzione. La durata della fase di esecuzione è senza dubbio breve confrontandola con i tempi di attesa.

10.7.3 Utilizzo di oggetti attivi

Come accennato nelle sezioni introduttive alle metodologie ad oggetti, un oggetto attivo possiede un suo flusso di esecuzione, cioè gli è associato un processo. Allora l'invocazione dei metodi tra gli oggetti può avvenire sfruttando meccanismi differenti.

1. L'oggetto mittente crea un nuovo processo per l'esecuzione del metodo dell'oggetto destinatario. In questo caso devono essere implementati meccanismi di protezione degli attributi dell'oggetto destinatario perché su di essi possono agire contemporaneamente i processi dei due oggetti coinvolti.

2. Ad ogni oggetto è associato un processo anche quando nessun metodo è in esecuzione. La comunicazione tra oggetti avviene mediante uno scambio di messaggi tra i processi corrispondenti che permette di individuare il metodo voluto che è eseguito dal processo dell'oggetto destinatario.

La seconda alternativa, sebbene meno efficiente, è senza dubbio più vicina ad una esecuzione multiprocessore ed ha il vantaggio di non richiedere l'uso di protezione sugli attributi degli oggetti. Ha, d'altro canto, lo svantaggio di richiedere meccanismi di comunicazione tra i processi e di imporre che tutti gli oggetti coinvolti nel modello siano attivi. Quest'ultimo fatto può portare ad un abbassamento delle prestazioni nell'attuazione dei processi software, perché in questo ambito molti oggetti hanno metodi brevi che possono essere più efficientemente invocati con un meccanismo simile ad una semplice chiamata a procedura, piuttosto che con uno scambio di messaggi tra processi.

Soluzione adottata dalla versione finale di S^3

S^3 utilizza una soluzione ibrida sfruttando i meccanismi messi a disposizione da Smalltalk per gestire la concorrenza. Gli oggetti i cui metodi hanno

esecuzioni piuttosto lunghe sono considerati come *attivi*. Essi non hanno un processo associato permanentemente, ma ogni volta che viene invocato un loro metodo, l'esecuzione di quest'ultimo è affidata ad un nuovo processo. Per quanto riguarda gli altri oggetti invece, l'esecuzione dei loro metodi è affidata al processo che realizza l'invocazione. Questo tipo di esecuzione concorrente può sembrare un po' complicato, ma è facilmente realizzabile mediante i meccanismi messi a disposizione da Smalltalk ed è descritto nel dettaglio nelle sezioni che trattano l'implementazione del modello dell'esempio di processo.

Estensione ad un ambiente distribuito

Volendo realizzare il progetto in un ambiente distribuito utilizzando oggetti attivi, si può pensare di avere una macchina diversa per ognuna delle persone che partecipano al processo software. Gli oggetti attivi che sono unicamente legati ad una certa persona sono eseguiti sulla macchina di tale persona; su una macchina centrale sono eseguiti i processi associati agli oggetti che sono utilizzati da tutti gli altri. In particolare si possono eseguire sulle macchine degli utenti le interfacce di comunicazione con le persone, gli strumenti automatici e gli oggetti che modellizzano le attività (istanze di `Task`) assegnate all'utente della 'work station'. Tutti gli oggetti devono invocare i metodi degli altri mediante l'invio di messaggi, o da una macchina all'altra oppure da un processo all'altro sulla stessa macchina.

Una tale soluzione presenta il vantaggio che, anche con processi software molto grandi, il carico di lavoro è uniformemente distribuito su tutto il sistema, ma presenta un notevole spreco di risorse dovuto allo scambio di messaggi, che è sicuramente più costoso della semplice invocazione di metodi. Inoltre la realizzzione di un tale ambiente richiede di avere a disposizione un supporto che consenta, non solo di far comunicare processi differenti, ma anche processi che si trovino su macchine diverse.

È stata presa in considerazione la possibilità di utilizzare gli ACA Service [D.E92a] per fornire il suddetto supporto e si sono valutate alcune possibilità per realizzare l'implementazione distribuita del modello dell'esempio di processo. Sebbene questa analisi non sia attinente agli argomenti di questa trattazione, vale la pena di sottolineare che è stata fatta basandosi sul principio che il modello di processo presentato è implementabile mediante qualsiasi strumento che permetta di gestire degli oggetti in grado di scambiare messaggi tra di loro. Quindi le possibili soluzioni sono tutte quelle che

mettono a disposizione, in modo più o meno distribuito, degli oggetti con degli attributi, dei metodi, e dei meccanismi di cominicazione tra di essi che permettono di far eseguire tali metodi.

11

Modellizzazione dei processi in S^3

La definizione di S^3, dello schema predefinito di classi e dei meccanismi che questo ambiente mette a disposizione per la modellizzazione dei processi, porta con sé l'introduzione di una particolare tecnica di modellizzazione dei processi software. Alla tecnica e a S^3 si associa anche un meta-processo che guida la realizzazione dei modelli di processo.

In questo capitolo sono date le specifiche per un processo software esemplificativo che non ha riscontro pratico, ma può essere utilizzato per fare valutazioni su S^3 e la tecnica di modellizzazione ad oggetti associata. Il modello di questo esempio di processo è anche riportato in questo capitolo, trattando ad una ad una le varie parti che lo compongono.

Infine è riportato un frammento di processo che richiede il potenziamento di alcuni dei meccanismi forniti da S^3 ed è proposta una soluzione al problema la quale mostra la flessibilità di S^3 e della sua tecnica di modellizzazione.

11.1 Modello di meta-processo

I concetti su cui si basa il progetto di S^3 consentono di introdurre un tecnica di modellizzazione ad oggetti dei processi software ed anche di un processo per realizzare ed eseguire (simulare nel caso di S^3) i modelli. Questo processo, dal momento che guida la modellizzazione dei processi software, è detto meta-processo ed il suo modello è schematizzato nella figura 11.1.

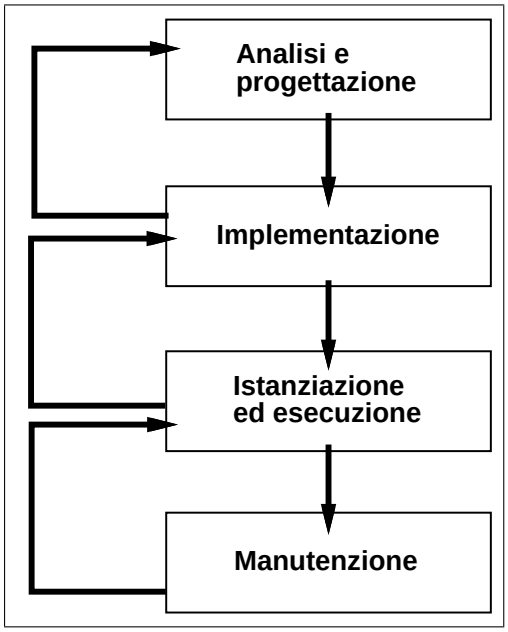

Figura 11.1. Modello di meta-processo

Nella prima fase si produce un modello di processo utilizzando un formalismo ad oggetti come quello della metodologia Coad/Yourdon descritto nel capitolo 6 e partendo dallo schema predefinito di classi fornito da S^3. Nella sezione 11.5 è presentato il prodotto di questa fase nella realizzazione del modello di un esempio di processo software. Questo modello è stato fatto utilizzando DECdesign, lo stesso strumento usato per progettare S^3.

Il progetto prodotto non è ad un livello di dettaglio tale da poter essere eseguito direttamente. Per questo su di esso deve agire una fase di `implementazione` che porta alla produzione di un modello *eseguibile* ('*enactable* model') del processo software. Tale descrizione è stata realizzata, per un esempio di processo software, utilizzando il linguaggio ad oggetti Smalltalk e partendo dalle classi predefinite forinite da S^3.

In ambienti futuri questa fase potrà essere automatizzata associando un'opportuna semantica ai vari elementi utilizzati per la modellizzazione del processo per i quali può essere largamente sfruttato il riutilizzo.

In questo lavoro la fase dell'`istanziazione ed esecuzione` del processo software è sostituita da una fase di `istanziazione e simulazione` che si basa su S^3. Questo permette di controllare la validità e la correttezza

del processo software realizzandone una esecuzione simulata. Le interfacce delle varie persone coinvolte nel processo sono eseguite tutte su una stessa macchina e gli strumenti necessari per la realizzazione del processo non sono realmente lanciati.

La fase della *menutenzione* non è stata affrontata in questo lavoro e consta nel valutare il modello realizzato per apportargli delle modifiche al fine di renderlo più efficiente.

Queste fasi non sono semplici passi sequenziali; come evidenziato dalla figura 11.1, l'esecuzione di ognuna di esse fornisce un maggiore grado di approfondimento nella conoscenza del problema che consente, ritornando alle fasi precedenti, di raffinarne il prodotto.

I principali sistemi di modellizzazione dei processi non distinguono tra una fase di `progettazione` ed una di `implementazione` del modello di processo. In essi la descrizione del processo è realizzata utilizzando un formalismo eseguibile.

S^3 dà supporto alla fase di `implementazione` ed a quella di `istanziazione ed esecuzione` grazie anche alle funzionalità messe a disposizione da Smalltalk. Il supporto alla fase di `analisi e progettazione` è dato da DECdesign in unione alle classi predefinite da S^3.

11.2 La tecnica di modellizzazione

S^3 è costituito da un insieme di classi che forniscono le funzionalità necessarie per la simulazione dei processi. La realizzazione di S^3 comporta anche l'introduzione di una tecnica di modellizzazione ad oggetti dei processi software. Tale tecnica prevede che un modello di processo sia realizzato come un progetto ad oggetti, in questo caso particolare sfruttando la metodologia Coad/Yourdon, e tutti gli aspetti del modello sono espressi mediante classi e relazioni tra esse. Un processo in esecuzione (nel caso di S^3 in simulazione) è costituito da un insieme di istanze delle classi del modello e di connessioni tra queste istanze.

I modelli di processo si realizzano creando specializzazioni delle classi generali che costituiscono S^3, oppure utilizzando direttamente le classi predefinite di utilità generale. Quindi questa tecnica di modellizzazione ad oggetti si basa su due punti di forza dell'orientamento agli oggetti, cioè l'*ereditarietà* ed il *riutilizzo*. In realtà anche l'ereditarietà si può vedere come una forma

di riutilizzo, che permette di trasmettere alle specializzazioni le funzionalità fornite dalla loro generalizzazione.

11.2.1 Riutilizzo dei meccanismi basato sull'ereditarietà

I meccanismi su cui si basa il funzionamento di S^3 sono sfruttati dalle classi tipiche del processo, ma sono forniti dalle classi più generali da cui queste utlime derivano per ereditarietà. Così le funzionalità fornite dalle classi che costituiscono S^3 vengono trasferite alle classi che compongono il modello, rendendo tale modello parte integrante dell'ambiente. Quando dalla progettazione del modello si passa alla sua implementazione, le classi del modello ereditano le funzionalità implementate per le classi di S^3 e quindi divengono esse stesse eseguibili. Ciò implica che non è l'ambiente che esegue (simula) i modelli, ma sono essi stessi che si eseguono.

Un processo può richiedere particolari funzionalità che S^3 non fornisce; in tal caso queste funzionalità possono essere fornite dalle classi stesse che costituiscono il modello. Ciò implica che le suddette funzionaltà devono essere esplicitamente implementate in queste classi e non possono essere ereditate a partire dalle classi di S^3. Se però altri processi necessitano di queste nuove funzionalità, il loro modello può essere creato utilizzando queste nuove classi o loro specializzazioni.

Nel progetto di S^3, sia la gestione gerarchica delle persone, sia quella degli strumenti automatici, sia la coordinazione tra le attività sono realizzate datta classe `Task`; allora qualsiasi sottoclasse di `Task` eredita tutti i suddetti meccanismi. Volendo invece dare la possibilità di riutilizzarli selettivamente senza doverli necessariamente ridefinire, si può pensare di non assegnare la realizzazione di questi meccanismi alla classe `Task`, ma di definire sottoclassi di quest'ultima ognuna delle quali sia responsabile di uno dei suddetti meccanismi (figura 11.2). Una tale soluzione non si è adottata nella progettazione di S^3 per non complicare troppo il modello di un ambiente prototipale, ma è importante per dare maggiore flessibilità al supporto per la modellizzazione dei processi fornito da un ambiente che si basi sui principi proposti in questa trattazione.

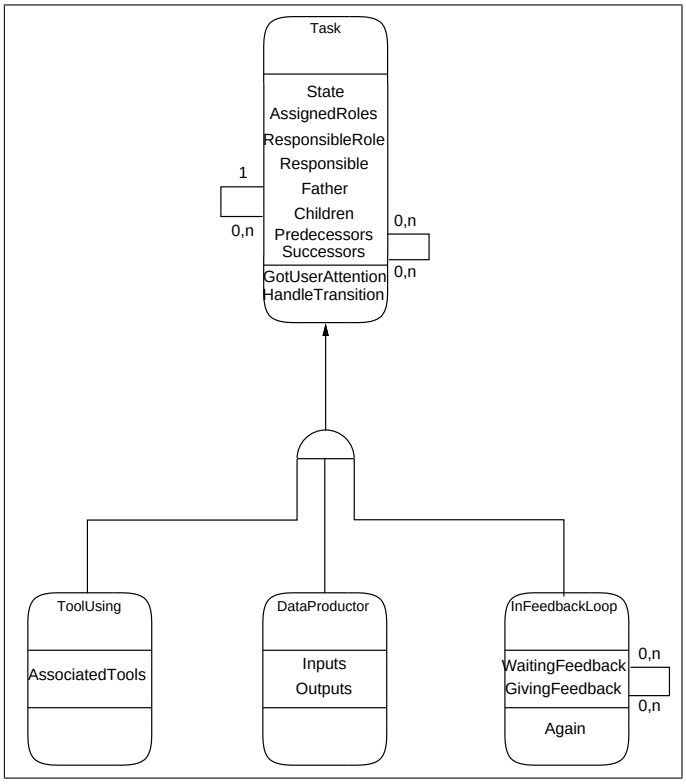

Figura 11.2. Sottoclassi speciali di `Task`

11.2.2 Riutilizzo delle classi definite per un processo

Come accennato nella sezione precedente, le classi definite per un certo processo, o in generale come supporto per la modellizzazione (cioè le classi che costituiscono S^3), possono essere utilizzate nella modellizzazione di nuovi processi facendo discendere da esse nuove classi mediante l'ereditarietà. È possibile però anche un tipo più diretto di riutilizzo, che consiste nell'includere nel modello di un nuovo processo classi che si erano create per altri modelli. Questa operazione richiede un minimo di accortezza perché è necessario conoscere bene le caratteristiche di queste classi e le connessioni di cui necessitano per poter funzionare nel modo corretto.

Ad esempio, è piuttosto probabile che nella modellizzazione di un nuovo processo possa essere utile un'attività di tipo `SelectTool`; però per utilizzare quella definita come supporto fornito da S^3, è necessario conoscere il modo

in cui opera per saperla posizionare correttamente nella gerarchia di decomposizione delle attività alfine di ottere da essa il risultato voluto. Oppure nel riutilizzare una classe che rappresenta un'attività come ad esempio la progettazione, è necessario sapere a quali tipi di dato di ingresso e di uscita debba essere connesa per funzionare correttamente, e quali connessioni di ordine di esecuzione richieda.

Il riutilizzo delle classi mette in evidenza il fatto che un buon ambiente per la modellizzazione dei processi software deve mettere a disposizione strumenti per generare e gestire librerie di classi predefinite di carattere molto generale, che sia possibile utilizzare nella modellizzazione dei vari processi. Inoltre è bene che i 'process model engineer' abbiano a disposizione degli strumenti che rendano il più semplice e sicuro possibile il riutilizo delle classi predefinite dal sistema e di quelle definite per la modellizzazione di altri processi, o addirittura di intere parti di modello.

11.2.3 Riutilizzo delle istanze

Nelle sezioni precedenti si è illustrato il riutilizzo di elementi a livello della modellizzazione dei processi, ma anche al momento dell'istanziazione del processo è possible fare uso di oggetti creati in precedenza.

Come accennato nella sezione 10.6 prima di cominciare l'attuazione di un modello di processo è necessario definire le risorse che questo avrà a disposizione. Ognuna di queste risorse è modellizzata da un oggetto che non deve essere necessariamente creato al momento dell'istanziazione, ma che potrebbe essere stato generato in precedenza, ad esempio perché utilizzato già in un altro processo. Allora l'ambiente deve fornire supporto:

- alla memorizzazione di tali oggetti in modo che la loro vita sia indipendente da quella del processo in cui sono utilizzati;
- al recupero degli oggetti che si ritengono utili in un processo e alla loro asegnazione ai processi in via di istanziazione.

Questi aspetti si sono trascurati nell'implementazione di S^3presentata nel seguito, perché si tratta di problemi non direttamente legati agli interessi di questo lavoro che sono diretti principalmente alla valutazione della bontà della tecnica di modellizzazione ad oggetti di processi software.

11.2.4 Flessibilità della tecnica di modellizzazione

Uno dei maggiori punti di forza di questa tecnica di modellizzazione ad oggetti di processi software, sta nel fatto che viene utilizzata una metologia di progettazione ad oggetti, come quella Coad/Yourdon, che è completamente generale ed, in quanto tale, utilizzabile per progettare programmi realizzabili con qualsiasi tipo di linguaggio ad oggetti. In realtà, secondo la teoria del modello ad oggetti, l'analisi e la progettazione ad oggetti possono essere utilizzate anche quando si voglia programmare con un linguaggio che non sia ad oggetti. In ogni caso il fatto che il modello sia così generale ed implementabile in modo piuttosto libero, permette di impostare nel modo che si preferisce l'esecuzione e di utilizzare i linguaggi o i supporti che meglio si adattano alle proprie esigenze.

Volendo però arrivare ad una implementazione ad oggetti del progetto realizzato, si può notare che non è strettamente necessario disporre di un linguaggio ad oggetti nel senso classico del termine, ma di un qualsiasi strumento informatico che metta a disposizione dei meccanismi per:

1. creare degli oggetti, con un loro stato (attributi) e dei metodi;
2. scambiare messaggi tra gli oggetti, che portino all'esecuzione dei loro metodi.

Un linguaggio di programmazione ad oggetti mette sicuramente a disposizione queste cose, ma esse possono anche essere ottenute da sistemi più complessi, che forniscono quindi altri vantaggi, quali basi di dati ad oggetti, ambienti per la comunicazione distribuita o anche ambienti con le caratteristiche volute realizzati all'uopo.

Il passaggio ad un ambiente distribuito è immediato perché il processo in esecuzione (simulazione nel caso di S^3) non è altro che un insieme di oggetti che comunicano tra loro. Allora per la realizzazione di un vero ambiente PM in ambito distribuito, si può pensare di utilizzare un linguaggio che metta a disposizione entità con le caratteristiche tipiche di classi ed oggetti le quali si trovano però su macchine differenti e comunicano tra loro anche da una macchina all'altra.

11.3 Un esempio di processo software

```
Si deve sviluppare un prodotto software avendo a disposizione
un gruppo di programmatori ed un responsabile del progetto
(‘project manager’).
   Tra i programmatori alcuni devono eseguire la progettazione
(‘design’), altri devono rivederlo (‘review’), altri devono
scrivere il codice.
   La revisione del progetto e la codifica possono essere
iniziate contemporaneamente, ma il processo non si può
considerare concluso fino a che il progetto non sia stato
approvato ed implementato.
   Lo strumento automatico di supporto alla progettazione deve
essere scelto dal responsabile durante l’esecuzione del
processo software.
   Lo strumento per la scrittura del codice va scelto tra
‘Emacs’ e ‘vi’ a seconda delle preferenze di chi lo deve
utilizzare.
   Il compilatore deve essere ‘CC’.
```

11.4 Caratteristiche del proceso esemplificativo

Intento di questo lavoro è la valutazione dell'efficenza del paradigma ad oggetti nel campo della modellizzazione dei processi software. A questo scopo si è realizzato un abiente per la simulazione dei processi software (S^3) ed a questo punto si vogliono applicare le tecniche proprie dell'analisi e della progettazione ad oggetti per realizzare il modello di un esempio di processo software. Quindi si vuole utilizzare Smalltalk-80 per implementare tale modello di processo per poi eseguirne una simulazione in S^3. Questo permette di valutare l'efficienza della tecnica proposta di modellizzazione ad oggetti dei processi software e di controllare il corretto funzionamento di S^3.

Sono noti diversi modelli di processo software standardizzati o anche più semplici problemi su cui basarsi per valutare le prestazioni di sistemi di modellizzazione dei processi, come gli esercizi proposti dal *6th International*

Software Process Workshop e dal *7th International Software Process Workshop*. Nonostante ciò si è deciso di formulare la definizione di un processo software ancora più semplice in modo da poterlo modellizzare, implementare e simulare focalizzando l'attenzione sugli strumenti utilizzati più che sul processo software in sé.

Sebbene sia molto semplice, il processo proposto come esempio, presenta i principali problemi che sorgono nella produzione del software:

- utilizzo degli strumenti automatici per produrre i documenti necessari;
- creazione della struttura del prodotto;
- mantenimento della coerenza della struttura del prodotto;
- decomposizione delle attività in sottoattività;
- assegnazione delle attività alle persone coinvolte nel processo;
- notifica delle assegnazioni ai responsabili;
- attivazione delle attività da parte dei responsabili;
- sincronizzazione e coordinazione delle attività;
- gestione del fallimento di attività.

Non vengono affrontate altre problematiche ugualmente importanti nella realizzazione di processi realmente utilizzabili per la produzione del software:

- durata delle varie attività;
- imposizione di tempi massimi di completamento;
- distribuzione bilanciata del carico di lavoro sulle persone coinvolte;
- condivisione delle risorse (che si considera gestita dal substrato su cui è costruito il sistema);
- pianificazione e soprattuto programmazione del processo ('*project management*');
- controllo dell'avanzamento del processo;

- manutenzione dopo la consegna del prodotto;
- evoluzione del modello del processo.

Dopo aver valutato se il paradigma ad oggetti può essere vantaggioso per modellizzare ed eseguire processi software, un futuro filone di ricerca potrà essere l'applicazione dei risultati ottenuti a modelli di processo software più completi.

11.5 Modello dell'esempio di processo

A partire dallo schema predefinito fornito da S^3, si è deciso di realizzare il modello dell'esempio di processo software proposto nella sezione 11.3. Questo modello è stato prodotto riutilizzando direttamente alcune delle classi messe a disposizione da S^3 (ad esempio quelle del soggetto `Role`) e creando specializzazioni di altre per modellizzare gli aspetti particolari del processo (ad esempio le attività).

Nelle sezioni seguenti sono riportate e descritte le principali parti del modello dell'esempio di processo. Non è stato possibile riportare in questa trattazione l'intero progetto del modello perché DECdesign non fornisce un formato di stampa utilizzabile in modo utile nella trattazione.

11.5.1 La gerarchia delle attività

La decomposizione gerarchica delle attività che intervengono nella modellizzazione dell'esempio di processo è riportata nella figura 11.3 (per rendere la figura più leggibile non si è riportato il livello della struttura, ma tutte queste classi sono specializzazioni di `Task`).

L'attività `DevelopProgram` è quella di più alto livello e consiste nello sviluppo del programma costituente il prodotto finale del processo. Questa attività si decompone in quattro sottoattività:

1. `SelectTool` serve per selezionare lo strumento automatico da utilizzare in un insieme di possibili scelte. Permette di soddisfare la richiesta dell'esempio proposto che lo strumento di sussidio alla progettazione sia scelto durante l'esecuzione del processo dal responsabile del processo. Le connessioni che interessano la classe `SelectTool` all'interno del modello saranno spigati nella sezione 11.5.5.

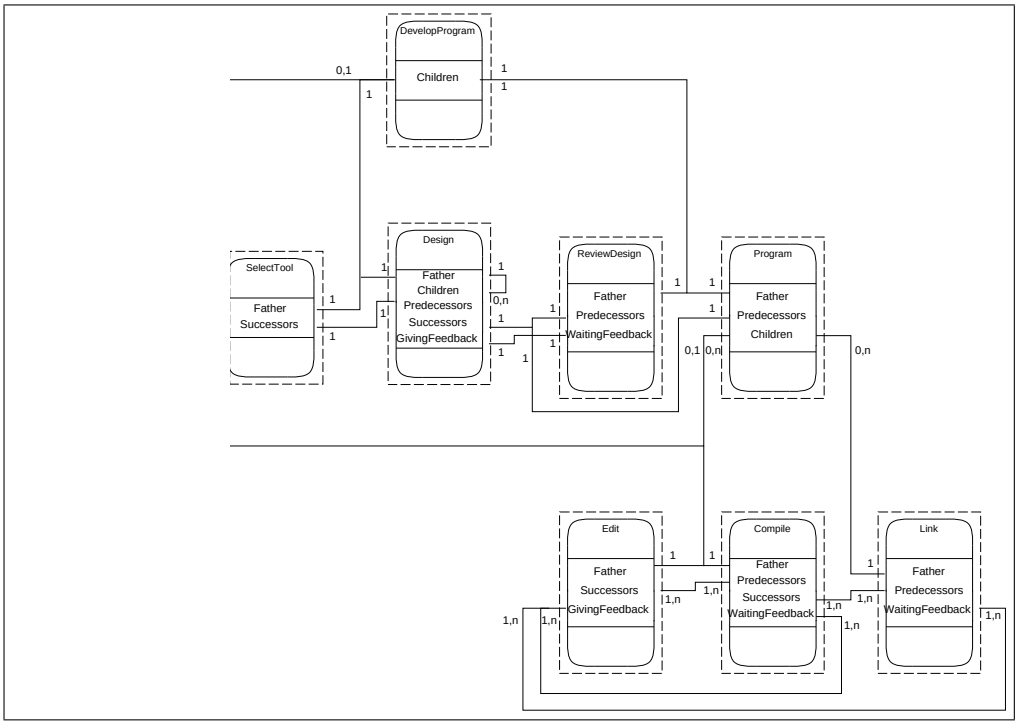

Figura 11.3. Le sottoclassi di `Task` nel modello del processo esemplificativo

2. `Design` è l'attività di progettazione del prodotto del processo. A sua volta si può decomporre in sottoattività della stessa classe, ma l'effettiva creazione di figli è una decisione che viene presa al momento dell'esecuzione del processo e non è imposta dal modello. Questo fatto è evidenziato dalla connessione tra `Father` e `Children` di istanze di `Design` la cui cardinalità (o,n) afferma che si ha possibilità di creare un numero arbitrario di sottoattività di progettazione.

3. `ReviewDesign` modellizza l'attività di revisione del progetto prodotto dalla progettazione.

4. `Program` rappresenta l'attività di condifica del programma e si può decomporre in tre tipi di sottoattività:

   (a) `Edit` consite nella scrittura dei vari sorgenti che compongono l'applicazione in via di sviluppo;

(b) `Compile` rappresenta la compilazione dei sorgenti;

(c) `Link` permette di costruire un eseguibile unendo i 'file' oggetto prodotti nella fase precedente.

11.5.2 I ruoli

Per la modellizzazione dell'esempio di processo sono sufficienti i ruoli forniti da S^3 che erano stati mostrati nella figura 9.5 e sono stati riportati nella figura 11.4 per maggiore comodità. Le sottoclassi di `Role` fornite da S^3 possono essere utilizzate direttamente, ma devono essere opportunamente connesse mediante `ResponsibleRole/CompetenceTasks` alle classi che rappresentano le attività per esprimere, a livello di modello, i ruoli che devono essere ricoperti dai responsabili.

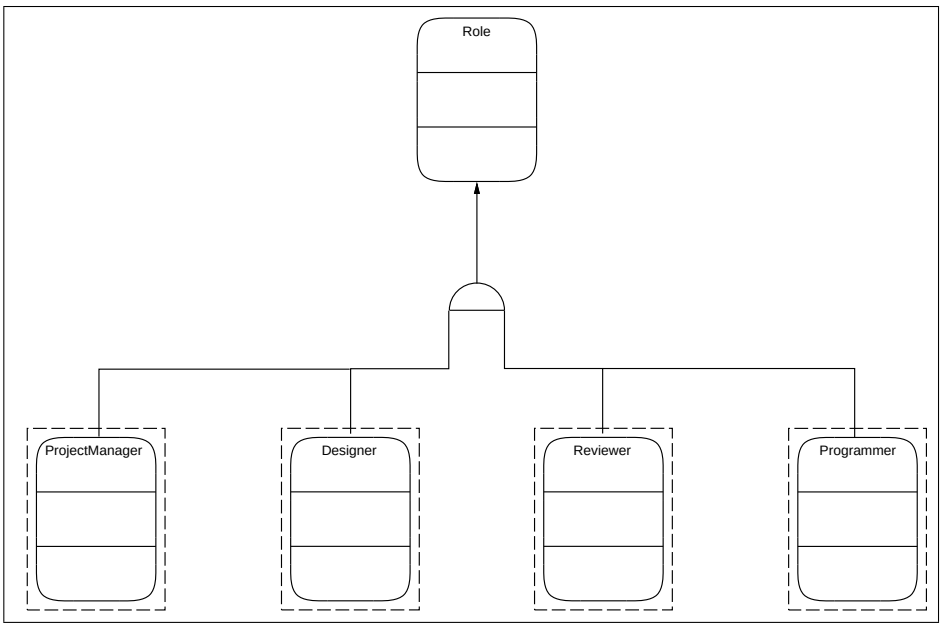

Figura 11.4. Il soggetto `Role` nel modello dell'esempio di processo

11.5.3 I dati

Per la modellizzazione dei dati prodotti dalle attività dell'esempio di processo software si è utilizzato il modello dei dati generico che è stato presentato nella

sezione 9.3.3. Tale modello si adatta perfettamente al software prodotto con il linguaggio C, come richiesto dalle specifiche del processo. Il soggetto `Data` è stato riportato nella figura 11.5.

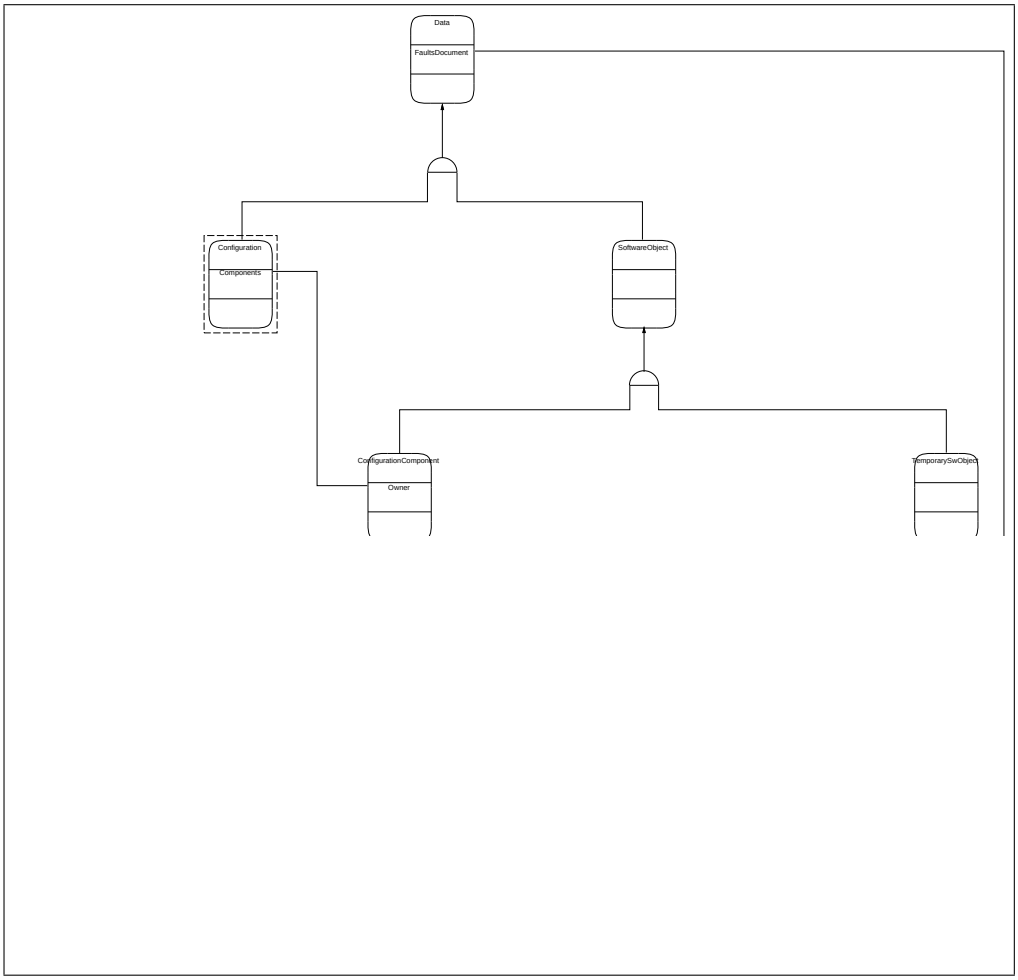

Figura 11.5. Il soggetto `Data` nel modello dell'esempio di processo

Le sottoclassi di `ConfigurationComponent` devono essere connesse mediante `Outputs/Productors` e `Input/Users` per esprimere, a livello di modello, quali tipi di attività producono ed utilizzano i vari tipi di dato.

11.5.4 Gli strumenti automatici

Per la modellizzazione degli strumenti automatici che sono necessari per la realizzazione del processo esemplificativo si sono create le sottoclassi di `Tool` riportate nella figura 11.6.

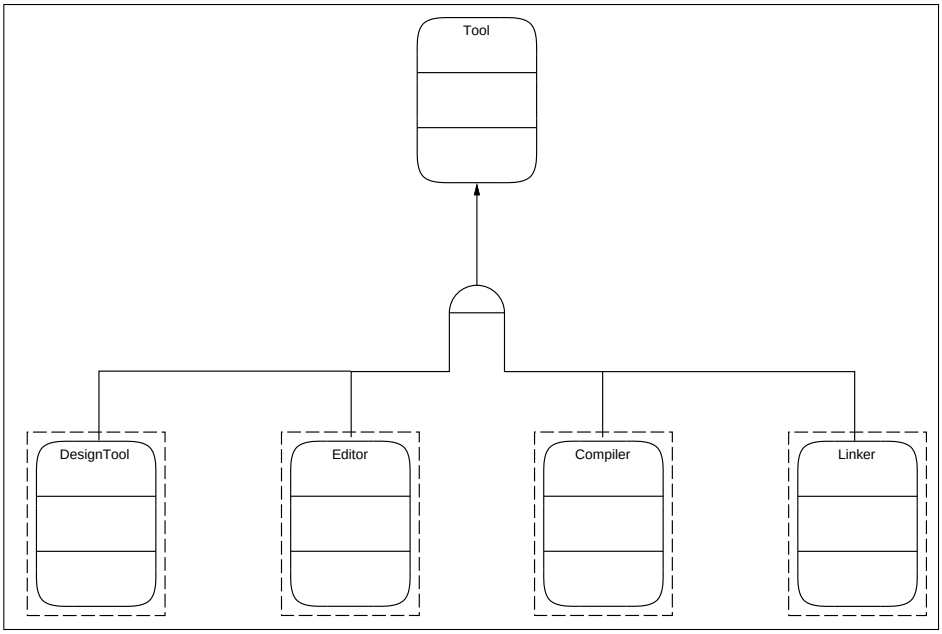

Figura 11.6. Il soggetto `Tool` nel modello dell'esempio di processo

Le sottoclassi di `Tool` devono essere collegate dalla connessione `Creator/Creator` alle sottoclassi di `Data` per definire a livello di modello i dati su cui questi strumenti possono operare.

11.5.5 `SelectTool` nel modello di processo

Nella figura 11.7 è riportata la classe `SelectTool` con le connessioni che essa richiede per la realizzazione del modello dell'esempio di processo. Nella sezione 9.4.2 è descritto il funzionamento delle attività modellizzate da `SelectTool`; tale comportamento permette di comprendere le modalità secondo cui la classe in questione è stata inserita nel modello.

Le specifiche del processo richiedono che durante l'esecuzione venga imposto l'uso di DECdesign come strumento per la progettazione. Questa specifica implica che un'istanza di `SelectTool` sia eseguita come componente di

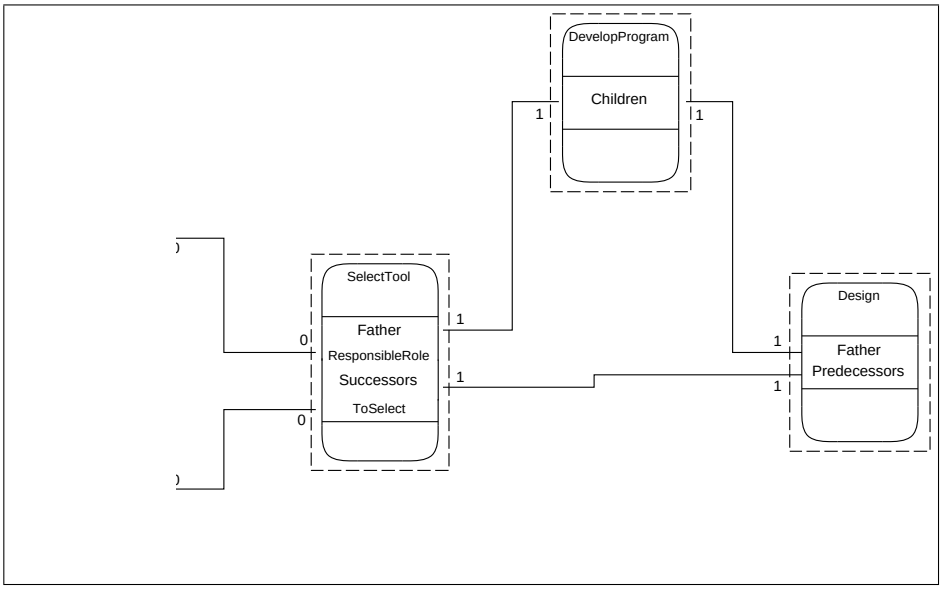

Figura 11.7. `SelectTool` nel modello dell'esempio di processo

un'attività che stia a più alto livello rispetto a `Design` nella gerarchia di decomposizione. Inoltre l'esecuzione di tale istanza di `SelectTool` deve essere terminata prima che una qualsiasi attività di `Design` sia istanziata. Allora, considerando la gerarchia di decomposizione riportata nella figura 11.3, l'unica possibilità è porre `SelectTool` allo stesso livello di `Design`, cioè come figlia di `DevelopProgram`. Questo è espresso dalla connessione tra l'attributo `Children` di `DevelopProgram` e `Father` di `SelectTool`.

All'attributo `ToSelect` è connessa la classe `DesignTool` perché la specifica dell'esempio di processo richiede che sia imposto l'uso dello strumento da utilizzare per la progettazione. La connessione `ResponsibleRole/CompetenceTasks` connette `ProjectManager` perché l'esempio di processo richiede che questa scelta sia operata dal responsabile del progetto. Infine la connessione `Successors/Predecessors` collega `Design` imponendo che questa attività, e quindi i suoi successori, non possano cominciare prima che `SelectTool` non sia terminata. Grazie al fatto che l'implementazione del modello utilizza l'istanziazione incrementale (spiegata nella sezione 10.2), nessuna istanza di `Design` è creata prima che possa essere eseguita, cioè prima che l'attività di selezione dello strumento non sia terminata.

11.6 Potenziamento dei meccanismi utilizzati nella modellizzazione dell'esempio di processo

Il processo software proposto nella sezione 11.3 rappresenta un caso piuttosto semplice e non ha probabilmente alcuna utilità pratica, però permette di fare utili considerazioni sull'efficienza della tecnica proposta per la modellizzazione ad oggetti dei processi software.

In questa trattazione si sono presentati gli strumenti ed i meccanismi offerti da S^3 che consentono di modellizzare l'esempio di processo; questi meccanismi sono stati pensati per avere valdità generale, ma in tutti i casi in cui la generalità portava ad un eccessivo grado di complicazione, si è preferito adottare una soluzione che andasse bene per casi non troppo particolari, ma fosse semplice, piuttosto che una sicuramente valida in generale, ma difficilmente comprensibile. Non è comunque complicato, anche in questi casi, accrescere le funzionalità offerte da S^3 o modificare lo schema di classi predefinito in modo che l'ambiente sia utilizzabile in casi particolari complicati.

Come esemplificazione di queste affermazioni riportiamo di seguito una possibile soluzione per rendere più generali e flessibili le relazioni di ordine tra le attività.

11.6.1 Relazioni di ordine più complicate

Nella progettazione di S^3 si è previsto un solo tipo di relazione di ordine tra le attività che è implementata mediante la connessione `Successors/Predecessors`. Il livello classe di questa connessione stabilisce un informazione utile per realizzare l'istanziazione incrementale delle nuove attività, mentre il livello istanza sta alla base dei meccanismi di coordinazione dell'esecuzione.

Il fatto che quest'unica relazione sia sufficiente nella modellizzazione dell'esempio di processo non significa che permetta di esprimere i vincoli di ordine tra le attività di un processo qualsiasi. Però non è difficile imporre in modo analogo, praticamente qualsiasi vincolo all'istanziazione ed all'esecuzione dei processi; ciò si può ottenere tracciando nuove connessioni ed aumentando i controlli durante l'evoluzione di un'attività attraverso i vari

stati.

In generale, con un meccanismo analogo a quello presentato nella sezione 10.1, è possibile vincolare il passaggio nello stato `S1` da parte di un'attività `T1`, al passaggio nello stato `S2` da parte di un'attività `T2`. Infatti la connessione `Successors/Predecessors` non è altro che un caso particolare di questa condizione, in cui lo stato `S2` è lo stato `Completed`, mentre lo stato `S1` è lo stato `Executing`. Inoltre il meccanismo dell'istaziazione incrementale fa sì che lo stato `Completed` di `T2` vincoli anche l'istanziazione di `T1`, ma in un caso più generale si dovrebbe avere una connessione di livello classe a parte, che esprima i vincoli sull'istanziazione.

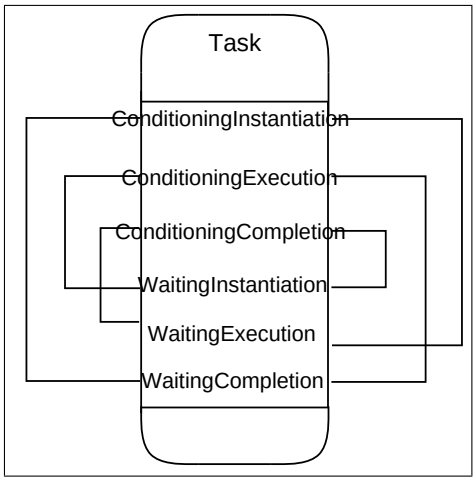

Figura 11.8. Classe `Task` generale e sue connessioni

Nella figura 11.8 è riportata una definizione della classe `Task` e di alcune delle connessioni tra le sue istanze (livello istanza) o tra le sue sottoclassi (livello classe) che permettono di realizzare particolari vincoli di ordine. La connessione `Successors/Predecessors` è sostituita da `Waiting-Completion/ConditioningExecution`; l'ordine tra le istanze collegate è ottenuto facendo in modo che quando un'attività deve passare nello stato `Executing` controlli che tutte le istanze connesse mediante l'attributo `ConditioningExecution` si trovino in uno stato tale da soddisfare la condizione dettata dalla connessione. In questo modo, ad esempio, l'esecuzione di un'attività può essere condizionata dal fatto che un'altra sia nello stato `Completed` ed un'altra ancora sia invece nello stato `Executing` o seguente.

In generale allora il vincolo che `T2` non possa passare nello stato `S2` fino a che `T1` non sia transitata nello stato `S1`, si può modellizzare mediante una connesione tra l'attributo `WaitingS1` di `T1` e l'attributo `ConditioningS2` di `T2`. Il rispetto delle condizioni può essere implementato sostanzialmente secondo due principi differenti.

1. Quando `T1` passa nello stato `S1` notifica del fatto tutte le istanze `Ti` connesse mediante l'attributo `WaitingS1` le quali memorizzano l'evento in modo da tenerne conto nei cambiamenti di stato interessati; cioè le transizioni che portano in stati `Sj` nel caso in cui si abbiano connessioni tra `WaitingS1` di `T1` e `ConditioningSj` di `Ti`.

2. Quando `T2` deve passare nello stato `S2` interroga tutte le attività `Ti` che gli sono connesse mediante l'attributo `ConditioningS2` per sapere se la condizione a loro legata è soddisfatta. Se `Ti` è connessa a `ConditioningS2` di `T2` mediante l'attributo `WaitingSj`, la condizione per il passaggio di stato legata a `Ti` è soddsfatta se `Ti` si trova in nello stato `Sj` o in uno successivo.

La prima soluzione è l'analogo di quella utilizzata nell'implementazione dell'esempio di processo per gestire la connessione `Successors/Predecessors`. Essa ha il vantaggio di minimizzare il numero di messaggi scambiati tra gli oggetti, ma può generare inconsistenza a causa della duplicazione di informazione che produce (vedi sezione 10.4).

11.6.2 Un esempio di vincoli complessi tra attività

Supponiamo di voler imporre tra le attività `Design`, `ReviewDesign` e `Program` i seguenti vincoli:

1. l'attività `ReviewDesign` non sia istanziata fino a che l'attività `Design` non è cominciata (questo ha senso solamente qualora si supponga che non sia utilizzata l'istanziazione incrementale);

2. l'attività `ReviewDesign` non sia eseguita fino a che `Design` non è in stato `Completed`;

3. l'attività `Program` non sia istanziata ed eseguita fino a che l'attività `Design` non sia in stato `Completed`;

4. l'attività `Program` non possa passare in stato `Completed` fino a che l'attività `ReviewDesign` non si trova in stato `Completed`.

Nella figura 11.9 è riportata una parte di modello che permette di imporre i vincoli sopraelencati; vediamo per ciascuno di essi, nel dettaglio, come viene imposto.

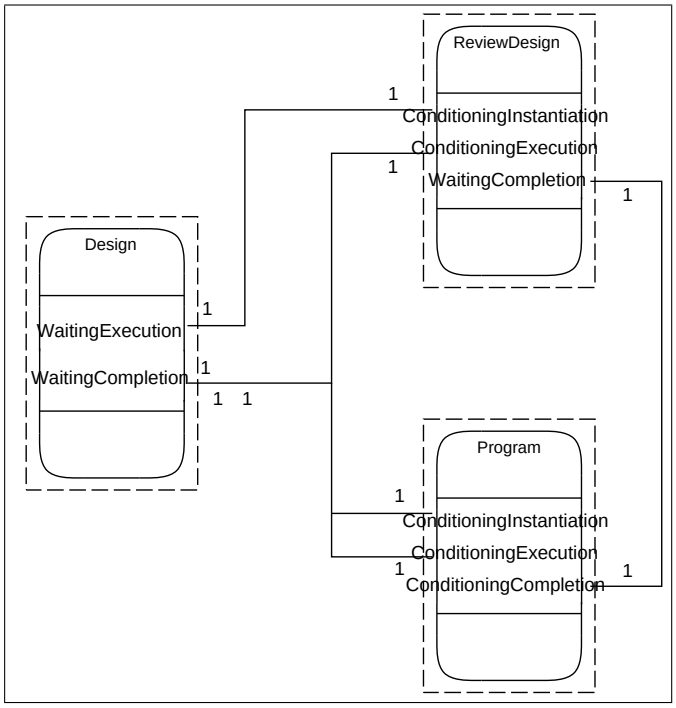

Figura 11.9. Esempio di vincoli complessi tra attività

1. È realizzato mediante la connessione tra l'attributo `WaitingExecution` della classe `Design` e l'attributo `ConditioningInstantiation` della classe `ReviewDesign`. Questa connessione ha significato a livello classe e deve essere utilizzata al momento dell'istanziazione degli oggetti di `ReviewDesign`.

2. È garantito dalla connessione tra `WaitingCompletion` di `Design` e `ConditioningExecution` di `ReviewDesign`.

3. Entrambe le connessioni che impongono queste condizioni hanno origine dall'attributo `WaitingCompletion` di `Design` e arrivano rispettivamente agli attributi `ConditioningInstantiation` e `Conditioning-Execution` di `Program`.

4. Questo vincolo è garantito mediante la connessione tra l'attributo `WaitingCompletion` della classe `ReviewDesign` e `ConditioningCompletion` di `Program`.

Parte IV

Implementazione ad oggetti di S^3

12

Aspetti generali dell'implementazione

In questo capitolo ha inizio la descrizione dell'implementazione di S^3, cioè delle classi che sono state definite per realizzarlo. Dopo aver fornito alcune informazioni di carattere generale, sono elencate le categorie in cui sono suddivise le classi che costituiscono S^3. Quindi si ha una descrizione di come sono stati gestiti la concorrenza e la protezione dei dati spiegando le classi ed i meccanismi realizzati per tale gestione. Inoltre è spiegato il supporto creato per la realizzazione delle connessioni che hanno un ruolo fondamentale nella tecnica di modellizzazione proposta. Infine è descritta una classe di utilità largamente utilizzata nell'implementazione di S^3.

12.1 Informazioni generali

L'ambiente S^3 è nato dalla necessità di ottenere un prototipo del progetto di ambiente fin qui descritto.

Struttura del codice

S^3 è implementato da un insieme di classi Smalltalk categorizzabili in due gruppi principali.

1. Classi che forniscono il supporto alla modellizzazione ed alla simulazione dei processi software e quindi costituiscono l'ambiente vero e proprio. Fanno parte di questo gruppo:

- le classi di utilità che permettono di implementare le connessioni e le interfacce di comunicazione con l'utente;
- le classi generali, come `Task`, da cui, per specializzazione, si possono derivare classi specifiche dei particolari processi; infatti come già detto nella sezione 11.2.1, queste classi forniscono il supporto per il funzionamento di meccanismi che sono propri dell'ambiente di modellizzazione e simulazione, non dei singoli processi.

2. le classi che implementano il modello ad oggetti dell'esempio di processo, e sono quindi tipiche di uno specifico processo, ma pur sempre utilizzabili in altri.

Poiché qualsiasi classe può essere riutilizzata per l'implementazione di nuovi processi, anche le classi create specificatamente per l'implementazione dell'esempio di processo entrano a far parte dell'ambiente stesso, come libreria di classi predefinite.

Notazione utilizzata

Per le classi già descritte nel progetto, nell'implementazione si sono utilizzati gli stessi nomi, tranne in casi particolari in cui questi coincidono con altri già esistenti. Nel modello, per gli attributi ed i metodi, si sono utilizzati nomi che iniziano con la lettera maiuscola, ma per le convenzioni imposte da Smalltalk-80, nell'implementazione devono cominciare con una lettera minuscola. Inoltre, poichè i selettori dei metodi hanno un numero diverso di parole chiave a seconda del numero di argomenti da passare al momento dell'invocazione, non sempre si è potuto utilizzare lo stesso nome presente nel progetto, ma si è cercato di mentenerli il più possibile simili.

Un'altra differenza nella notazione, sta nel modo di indicare gli stati delle attività. Nel progetto si sono utilizzati nomi che iniziano con la lettera maiuscola; nell'implementazione si sono utilizzati simboli costituiti dalla stessa sequenza di caratteri, ma ovviamente comincianti con il carattere `#`.

Per uniformità con il codice riportato nelle figure dei capitoli riguardanti l'implementazione di S^3 e nell'appendice C, da questo punto in poi si utilizza la notazione usata nell'implementazione, anche quando si fa riferimento ad elementi del progetto.

12.2 'Class category' di S^3

Le classi create per l'implementazione di S^3 sono state raggruppate in categorie. Alcune di esse coincidono con i soggetti in cui è suddiviso il modello dell'esempio di processo realizzato con la metodologia Coad/Yourdon: `Task`, `Role`, `Data` e `Tool`. Le seguenti, invece, non sono in diretta relazione con i soggetti del progetto.

Persons Contiene la classe `Person` individuata nel modello del processo che modellizza le persone coinvolte. Nel progetto ad oggetti questa classe non è parte di nessun soggetto, ma Smalltalk-80 impone che qualsiasi classe sia inclusa in una categoria.

Relations Le lassi di questa categoria implementano le connessioni, cioè le relazioni tra gli oggetti. I metodi invocabili sulle istanze di queste classi permettono di interrogare queste relazioni nel modo più adeguato al ruolo che esse ricoprono nel sistema.

Utilities Si tratta di una serie di classi i cui oggetti sono utilizzati in situazioni generali e non correlate tra loro.

User interfaces Contiene le classi che è stato necessario creare per l'implementazione dell'interfaccia utente. Le istanze di queste classi sono per lo più utilizzate come 'model' nell'ambito della struttura di Smalltalk- 80 per la creazione e gestione di interfacce grafiche, come spiegato nell'appendice B.7.1.

12.3 I flussi di esecuzione

Nella sezione 10.7 sono state proposte alcune possibili alternative per l'utilizzo di uno o più flussi di esecuzione ('control thread') nell'implementazione di S^3. La gestione dei flussi di esecuzione che S^3 realizza si ripercuote sul grado di parallelismo della simulazione dei modelli di processo.

Una prima versione di S^3 è stata realizzata utilizzando un flusso di esecuzione per la gestione di ogni interfaccia di interazione con l'utente. Quando l'utente intraprende qualche azione, vengono invocati i metodi degli oggetti `Task` da parte dell'interfaccia mediante cui la persona ha comunicato e tali

metodi sono eseguiti dal flusso di esecuzione proveniente dall'interfaccia stessa. Questa impostazione è piuttosto semplice perché non richiede un grande numero di accorgimenti per la gestione della condivisione dei dati da parte di processi concorrenti, però è piuttosto banale, inefficiente e soprattutto non accettabile quando si debbano attuare o simulare processi software di grosse dimensioni.

12.3.1 Gli oggetti attivi in S^3

La versione di S^3 presentata in questa trattazione è leggermente più complessa, ed utilizza un discreto numero di processi concorrenti considerando come oggetti attivi non solo le interfacce, ma anche le istanze delle sottoclassi di `Task`.

Objectworks(r)/Smalltalk non prevede un particolare supporto per l'implementazione di oggetti attivi e la loro comunicazione, ma fornisce i potenti meccanismi descritti nell'appendice B.8 per la creazione e gestione di processi concorrenti. Gli oggetti attivi sono stati implementati creando un nuovo processo per l'esecuzione dei loro metodi ogni volta che sono invocati da parte di un'altro oggetto. La creazione del nuovo processo per l'esecuzione di un metodo può avvenire in due modi:

1. il messaggio viene incluso in un blocco su cui si invoca il metodo `fork`;
2. il codice del metodo consiste di un unico blocco su cui è invocato il metodo `fork`.

La seconda soluzione è più coerente con i concetti alla base dell'orientamento agli oggetti in quanto il fatto che l'oggetto abbia un suo flusso di esecuzione è ottenuto mediante l'implementazione dell'oggetto stesso. Invece nella prima soluzione è il modo di invocare i metodi, e quindi l'implementazione del chiamante, che determina il tipo di funzionamento del chiamato. Nonostante ciò la soluzione adottata nella realizzazione di S^3 è la prima, per il fatto che molte invocazioni di metodi sulle istanze della classe `Task`, cioè una delle categorie di oggetti che si considerano attivi, sono finalizzate ad ottenere una risposta che dovrebbe in ogni caso essere attesa dal mittente. Allora, per non complicare troppo la comunicazione tra gli oggetti, si è scelta una soluzione ibrida che realizza l'invocazione dei metodi con un nuovo processo solo quando questo è realmente utile.

Per come si è realizzata la gestione dei flussi di esecuzione in S^3, il processo associato ad ogni oggetto attivo cessa di esistere quando termina l'esecuzione del metodo la cui invocazione ha provocato la creazione del processo in questione. Dunque in reltà i processi sono associati ai metodi in esecuzione, e non agli oggetti; allora un oggetto può contenere più di un flusso di esecuzione se riceve due invocazioni contemporanee a suoi metodi.

La scelta di utilizzare oggetti attivi per l'implementazione di interfacce utente ed attività, è stata dettata dal fatto che la durata delle operazioni che questi compiono è piuttosto lunga e spesso non è fissa, soprattutto per quanto riguarda le interfacce che devono attendere l'interazione della persona interessata. Per gli altri oggetti invece è sufficiente utilizzare normali invocazioni di metodi, la cui esecuzione è assegnata al processo che sta eseguendo il metodo che ha inviato il messaggio.

12.3.2 Protezione degli attributi

In presenza di più flussi di esecuzione è necessario assicurare una corretta condivisione dei dati a cui i vari processi hanno accesso. Questi dati sono gli attributi degli oggetti, e non solamente quelli attivi, ma anche gli altri che Booch chiama oggetti *bloccanti* [Boo91]. All'interno di questi oggetti si possono trovare più flussi di esecuzione provenienti da altrettanti oggetti attivi che ne hanno invocato i metodi.

Il problema della condivisione dei dati si traduce nella necessità di accedere in modo protetto agli attributi di molti degli oggetti presenti nel sistema. Non è necessario assicurare questa protezione a tutti gli oggetti presenti nel sistema perché alcuni ricevono messaggi solo da un particolare oggetto e quindi non possono venire ad avere due processi che eseguono i loro metodi. La protezione è sicuramente necessaria per gli oggetti attivi e per le istanze delle classi definite nel modello del processo che si vuole simulare.

In generale per proteggere una o più variabili condivise da processi differenti, è necessario che le istruzioni che accedono ad esse siano contenute in regioni critiche associate ad uno stesso semaforo. A partire da questo presupposto di possono considerare alcune possibili gestioni della condivisione dei dati in S^3.

- Ad ogni variabile si associa un semaforo per controllare l'accesso alle regioni critiche contenenti le istruzioni che la manipolano. Questa soluzione, sebbene sia la più efficiente dal punto di vista del tempo per

cui i vari processi restano bloccati, ha due notevoli svantaggi:

1. richiede l'utilizzo di un gran numero di semafori;
2. può facilmente portare a delle situazioni di attesa ciclica da parte dei processi ('*deadlock*').

- Si individuano insiemi di variabili che vengono sempre accedute in gruppo o caratterizzate dal fatto che, quando se ne utilizza una, se si blocca l'accesso anche alle altre, il sistema non risulta rallentato. Quindi si associa ad ogni insieme di variabili un semaforo che controlla le regioni critiche in cui avvengono gli accessi alle variabili dell'insieme in questione.

- Si associa un semaforo ad ogni oggetto e si includono in regioni critiche da esso controllate tutte le istruzioni che accedono agli attributi dell'oggetto. Questa soluzione è palesemente un caso particolare della precedente in cui gli insiemi di variabili sono costituiti dagli attributi di un singolo oggetto.

La gestione realizzata in S^3 utilizza un meccanismo molto vicino a quello di *monitor* per gli oggetti attivi, e l'ultima soluzione proposta, per i metodi degli altri oggetti. Il 'monitor' è usato diffusamente nei sistemi concorrenti ed è un modulo di codice che può essere eseguito da un solo processo alla volta. Questa è proprio la caratteristica tipica della regione critica, ma l'uso è leggermente differente e soprattutto l'implementazione è trasparente ed indipendente dal modo in cui si accede al 'monitor'. Infatti il processo che invoca le funzioni del 'monitor', lo fa senza utilizzare un particolare protocollo, ed è il 'monitor' a causare un eventuale accodamento del chiamante ed a risvegliarlo quando l'esecuzione può avere luogo. La coda di attesa ed il risveglio possono essere gestiti secondo politiche differenti, ma ciò resta pur sempre responsabilità unica del 'monitor'.

I metodi degli oggetti attivi vengono divisi tra quelli *pubblici*, cioè invocabili da parte di altri oggetti, e quelli strettamente *privati*, cioè invocati solo da parte di altri metodi dell'oggetto stesso. Quindi il codice dei metodi pubblici viene completamente incluso in una regione critica ottenendo che un solo metodo alla volta può essere eseguito su ogni oggetto così che i vari attributi non possono essere manipolati contemporaneamente da più processi. L'oggetto su cui è invocato il metodo è allora come un 'monitor' perché

l'invocazione è come tutte le altre, e chi la fa non agisce in modo particolare, ma potrebbe restare bloccato sulla chiamata. È l'oggetto destinatario che si occupa della gestione del bloccaggio e del successivo risveglio utilizzando una semplice politica FIFO ('First In First Out'). I metodi privati, essendo invocati all'interno della regione critica di quelli pubblici, non necessitano chiaramente di essere racchiusi in regione critica[1].

12.3.3 Attese cicliche

La soluzione proposta richiede un numero ragionevole di semafori, per il fatto che ne è sufficiente uno per ogni oggetto, e non richiede onerosi accorgimenti per evitare attese cicliche.

Supponiamo che un oggetto `O1` si blocchi invocando un metodo dell'oggetto `O2`, perché su `O2` è già in esecuzione un metodo invocato da `O3`; se questo metodo a sua volta manda un messaggio ad `O1` si ha un 'deadlock'. Una situazione di questo genere sta ad indicare che c'è una conoscenza reciproca di `O1` ed `O2`; questa è un'eventualità piuttosto remota in S^3 in cui, come si evidenzia trattando dell'implementazione delle varie classi, le connessioni sono quasi tutte monodirezionali. A questo va sommato il fatto che in S^3 i flussi di controllo possono provenire solamente dalle interfacce o dalle attività.

Nei rari casi in cui si può avere una situazione critica del tipo descritto, si deve evitare di generare un ciclo di attesa liberando l'accesso ad `O1`, cioè simulando l'uscita dalla regione critica, mediante l'invocazione del metodo `signal` sul semaforo associato, prima dell'invio del messaggio ad `O2`. Per assicurare il corretto funzionamento, è altresì necessario che il primo messaggio dopo quello ad `O2` sia l'invocazione del metodo `wait` sul semaforo associato ad `O1` per bloccare nuovamente la regione critica.

Nell'analizzare l'implementazione dell'esempio di processo si può vedere una serie di situazioni in cui è esemplificato questo meccanismo. Un caso particolare è quello dell'attesa di una risposta da parte del responsabile di un'attività che è gestita come descritto nella sezione 13.3.7.

[1] Diversamente di genererebbe un 'deadlock', perché al momento dell'invocazione la regione critica sarebbe già occupata.

12.3.4 La classe `Controlled`

La classe `Controlled`, la cui definizione è riportata nella figura 12.1, non ha istanze, ma da essa devono derivare per ereditarietà tutte le classi i cui oggetti necessitano di protezione sugli attributi.

```
Object subclass:  #Controlled
   instanceVariableNames:  'semaphore '
   classVariableNames:  ''
   poolDictionaries:  ''
   category:  'Utilities'
```

Figura 12.1. Definizione della classe `Controlled`

Creazione delle istanze

Il metodo `new` per la creazione delle istanze fornito dalla superclasse `Object`, è stato ridefinito. L'implementazione del nuovo metodo `new` riportata nella figura 12.2, mostra che il nuovo oggetto è creato invocando il metodo standard di istanziazione, definito per la classe `Object`, e quindi all'istanza da esso restituita è inviato il messaggio `thisInitialize`; il valore da questo ritornato è restituito come argomento di ritorno di `new`.

```
new
   "create a new instance"

   ↑super new thisInitialize
```

Figura 12.2. Metodo `new` della classe `Controlled`

Nella figura 12.3 è riportata l'implementazione del metodo di inizializzazione delle istanze della classe `Controlled` che non fa altro che assegnare all'attributo `semaphore` una nuova istanza di `Semaphore` creata invocando sulla classe `Semaphore` il metodo `forMutualExclusion`.

La ridefinizione del metodo `new` non sarebbe indispensabile perchè si potrebbe lasciare il compito dell'invocazione del corrispondente metodo di `Object` al meccanismo di selezione di Smalltalk che, non trovando una definizione di `new` in `Controlled`, la cercherebbe tra le sue superclassi. In tal

```
thisInitialize
    semaphore := Semaphore forMutualExclusion
```

Figura 12.3. Metodo `thisInitialize` della classe `Controlled`

caso le sottoclassi di `Controlled` dovrebbero provvedere all'inizializzazione dell'unico attributo `Semaphore` ad esempio invocando metodo di inizializzazione definito per `Controlled`.

La ridefinizione del metodo `new` nel modo presentato nella figura 12.2 permette proprio di sollevare le sottoclassi dall'onere di invocare il metodo di inizializzazione di `Controlled`. Questo modo di implementare i metodi, in perfetto accordo con i principi dell'orientamento agli oggetti, conferisce maggiore robustezza perché un'errata inizializzazione di `semaphore` causerebbe il mancato funzionamento di tutti i meccanismi di protezione dei dati descritti in precedenza.

Alcune considerazioni generali

A proposito dei metodi appena mostrati si possono fare delle considerazioni che permettono di capire meglio alcuni dei meccanismi utilizzati da Smalltalk descritti nell'appendice B.

Inizializzazione delle variabili di istanza Sebbene l'inizializzazione degli oggetti `Controlled` sia molto semplice e richieda una sola assegnazione, si è definito un metodo appositamente per questo. Questo fatto non è una semplice questione di chiarezza, ma è imposto dal fatto che l'inizializzazione delle istanze non può essere fatta all'interno del metodo responsabile della loro creazione. Infatti il metodo `new` è di livello classe per cui durante la sua esecuzione esso non può accedere agli attributi dell'oggetto che ha appena creato per assegnare loro un valore. Quindi ogni classe deve disporre di:

1. un metodo di livello classe per la creazione delle istanze, che può essere eventualmente ereditato dalla superclasse;
2. un metodo di livello istanza per l'inizializzazione dei nuovi oggetti, che deve essere assolutamente ridefinito qualora la classe in questione abbia degli attributi non presenti nella superclasse oppure che debbono essere inizializzati in modo differente.

Selezione dei metodi Normalemente, come si vede nel resto della descrizione dell'implementazione di S[3], il selettore del metodo di inizializzazione delle istanze è sempre `initialize`, mentre in questo caso si è utilizzato `thisInitialize`. Questa scelta nasce da una necessità imposta dal meccanismo di selezione dei metodi implementato da Smalltalk-80. Qualsiasi classe demanda la creazione delle proprie istanze al metodo `new` della propria superclasse; ciò è fatto anche da `Controlled` che invoca la `new` di `Object`. Quando viene mandato il messaggio `new` su una sottoclasse di `Controlled`, o per invocazione nell'implementazione di quest'ultimo, o per il meccanismo di selezione dei metodi se non è stato ridefinito, viene eseguito il codice riportato nella figura 12.2. Quando avviene l'invio del messaggio `thisInitialize` il meccanismo di selezione dei metodi comincia a cercarlo a partire dall'oggetto che sta eseguendo il metodo che nel caso in questione è una sottoclasse di `Controlled`. Se tale classe dispone del metodo `thisInitialize`, questo viene eseguito invece di quello riportato nella figura 12.3 definito per la classe `Controlled`. Poichè con alta probabilità la ogni classe definisce un metodo di inizializzazione delle proprie istanze identificato mediante il selettore standard `initialize`, se lo si fosse utilizzato anche per il metodo di `Controlled` questo non sarebbe stato invocato al momento dell'esecuzione della `new`, ma sarebbe stato eseguito il codice del metodo omonimo della sottoclasse direttamente interesata dalla creazione di istanze[2].

Valori di ritorno dei metodi Convenzionalmente il valore di ritorno del metodo `new` di creazione delle istanze è l'istanza stessa. Nell'implementazione riportata nella figura 12.2 viene invocato il metodo `new` della superclasse che restituisce il nuovo oggetto, ed a quest'ultimo è inviato il messaggio `thisInitialize`. Il valore ottenuto da questa esecuzione è utilizzato come valore di ritorno del metodo mediante l'operatore ↑; allora perché il metodo `new` così definito restituisca la nuova istanza, bisogna che questo sia il valore di ritorno di `thisInitialize`, mentre nella figura 12.3 non è specificato alcun argomento di ritorno mediante l'operatore ↑. Tutto ciò è appropriato e dà origine ad un funzionamento corretto, perché, come affermato nella sezione B.5.2, se non è specificato diversamente, ogni metodo restituisce l'oggetto su cui è eseguito; nel caso in esame tale oggetto è proprio la nuova istanza di

[2]Poichè `Controlled` è un tipico caso di classe astratta, un metodo `initialize` definito per controlled non sarebbe mai utilizzato a meno che in uno dei metodi delle sue sottoclassi dirette non comparisse un messaggio `super initialize`.

`Controlled`.

Metodi per la gestione dei flussi di esecuzione

Gli oggetti della classe `Controlled` hanno la sola caratteristica di essere associati ad un semaforo che deve essere utilizzato per la gestione dei processi che eseguono i metodi in modo da garantire la protezione degli attributi. Sono disponibili i messaggi:

- `critical:`
- `signal`
- `wait`

I metodi ad essi associati non fanno altro che inviare gli omonimi mesaggi al semaforo individuato dalla 'instance variable' `semaphore`. Grazie alla definizione di questi messaggi, la classe `Controlled` presenta la caratteristica che i messaggi da inviare per garantire la sicurezza dei dati vengono indirizzati all'oggetto stesso. Come si vede nel seguito della trattazione, tutte le classi che necessitano di protezione sui dati sono derivate per ereditarietà dalla classe `Controlled`.

Con l'approccio utilizzato, sfruttando una delle caratteristiche fondamentali del modello ad oggetti quale l'incapsulamento, il meccanismo di protezione funziona in modo del tutto indipendente da come è effettivamente stato implementato; infatti l'inizializzazione del semaforo associato e la comunicazione con esso avvengono solamente nei metodi della classe `Controlled`. Si potrebbe cambiare completamente l'implementazione di questa classe e, a patto di fornire degli equivalenti metodi `critical:`, `wait` e `signal` , il resto del sistema funzionerebbe correttamente e senza dover subire modifiche.

12.4 Le connessioni

Dalla presentazione del modello dell'esempio di processo si può facilmente comprendere che le connessioni rivestono una notevole importanza in questa tecnica di modellizzazione dei processi.

La connessione tra due oggetti può essere in generale tradotta nel fatto che ognuno dei due possiede un riferimento all'altro così che ne può invocare

i metodi. In Smalltalk questo si può tradurre nel fatto che ognuno dei due oggetti abbia un attributo cui è assegnato l'identificatore dell'altro oggetto.

Per la modellizzazione dei processi è vantaggiosa un'implementazione più efficiente e flessibile che permetta di connettere ad uno stesso oggetto molte istanze, anche di classi diverse, e di accederle in modo differenziato a seconda del significato della connessione implementata. Proprio a questo fine sono state create le classi della categoria `Relations`.

12.4.1 Connessioni di livello istanza

In generale una connessione di livello istanza mette in relazione due oggetti, cioè fornisce ad ognuno un riferimento per l'altro. Però un oggetto può essere collegato a molte istanze dalla stessa connessione, quindi abbiamo scelto di utilizzare un oggetto della classe `Set` per implementare il tipo più generale di connesione di livello istanza.

Nel progetto del modello di processo si è assunto che le 'instance connection' rappresentino un legame bidirezionale sia tra le classi collegate che tra le loro istanze come detto nella sezione 6.5. In realtà non è detto che sia sempre utile che ciascuno dei due oggetti connessi abbia un riferimento all'altro.

Nel seguito sono descritte alcune delle classi i cui oggetti sono utilizzati per implementare connessioni sia di livello classe che di livello istanza, ma il loro utilizzo viene spiegato trattando della classe `Task`.

Classe `InstanceConnection`

Se le istanze di una classe hanno necessità di fare riferimento ad altri oggetti, si deve definire per questa classe un attributo per ogni connessione. Quindi le istanze, quando sono create, provvedono ad assegnare ad ognuno di questi attributi un'istanza della classe `InstanceConnection` la cui definizione è riportata nella figura 12.4. Nell'implementazione del modello dell'esempio di processo per l'attributo che permette di realizzare la connessione, si è utilizzato lo stesso nome usato nel progetto, con le differenze di notazione descritte nella sezione 12.1.

Per la classe `InstanceConnection` sono stati definiti un insieme di metodi per collegare oggetti interessati mediante la connessione (ad esempio `add:`) e per interrogare le connessioni nel modo più utile a seconda dell'occorrenza (ad esempio `instancesOfClass:`).

```
Set variableSubclass:  #InstanceConnection
   instanceVariableNames:  ''
   classVariableNames:  ''
   poolDictionaries:  ''
   category:  'Relations'
```

Figura 12.4. Definizione della classe `InstanceConnection`

Nella figura 12.5 è riportato il codice del metodo `instancesOfClass:` che restituisce un `Set` contenente tutti gli oggetti, collegati mediante la connessione su cui è invocato il metodo, che appartengono alla classe passata come argomento. Quindi se `connection` è una variabile cui è stato assegnato un oggetto `InstanceConnection`, volendo conoscere tutte le istanze della classe `Design` collegate, si invia il messaggio:

```
connection instancesOfClass:  Design
```

Vediamo nel dettaglio come viene implementato il metodo riportato nella figura 12.5. Poiché si deve ritornare un sottoinsieme degli oggetti collegati mediante la connessione interessata, in primo luogo viene creato l'oggetto che verrà restituito come valore di ritorno invocando il metodo `new` della classe `Set`. Il nuovo oggetto creato è assegnato alla variabile temporanea `tmp`. Quindi viene invocato il metodo `do:` sulla connessione (identificata da `self`): questo metodo viene ereditato dalla classe `Set` e permette di eseguire il blocco che gli è passato come argomento per ognuno degli elementi contenuti nell'insieme. L'elemento su cui si sta eseguendo il blocco è passato a quest'ultimo e viene assegnato di volta in volta al parametro `related`.

Si ricava la classe dell'elemento in esame inviando a quest'ultimo il messaggio `class` e quindi la si confronta con l'argomento della chiamata `classOfRelated` utilizzando il messaggio binario `=` che restituisce un'istanza di `Boolean` su cui viene invocato il metodo `ifTrue:`. L'inserimento dell'elemento in `tmp` è realizzato invocando su quest'ultimo il messaggio `add:`.

Classe `BooleanInstanceConnection`

È utile la creazione della classe `BooleanInstanceRelation`, le cui istanze permettono di implementare un tipo particolare di relazione in cui, per ogni oggetto collegato, è memorizzato un valore booleano. Questa classe può essere utilizzata per casi più generali, ma in particolare è utile ed efficiente

```
instancesOfClass: classOfRelated
   "returns the set of instances of class childClass"

   | tmp |
   tmp := Set new.
   self do:  [:related | related class = classOfRelated ifTrue:
[tmp add:  related]].
   ↑tmp
```

Figura 12.5. Implementazione del metodo `instancesOfClass:`

nell'implementazione della connessione che lega ad un'attività quelle che le precedono, cioè il valore dell'attributo `predecessors` della classe `Task`, oppure nella realizzazione del collegamento delle sottoattività, cioè il valore dell'attributo `children` (sezione 13.3.2).

Ci sono casi in cui per ognuno degli oggetti collegati dalla connessione è necessario mantenere altre informazioni oltre ad un semplice riferimento ad essi; un tipico esempio è rappresentato dalle connessioni di precedenza `Successors/Predecessors`. Nella sezione 10.4 sono stati proposti alcuni meccanismi di scambio di messaggi per la coordinazione delle attività, e si è accennato che nell'implementazione di S^3 si è deciso di fare in modo che quando un oggetto `Task` cambia il proprio stato, lo comunica a tutte le istanze che gli sono connesse mediante l'attributo `successors`. Queste memorizzano l'evento e, quando devono controllare se è possibile cambiare stato, non interpellano più gli oggetti connessi mediante `predecessors`. Da questi accenni sul funzionamento del meccanismo si possono fare alcune deduzioni:

1. è sufficiente memorizzare, per ognuno degli oggetti connessi mediante `predecessors`, se la condizione ad esso associata è soddisfatta oppure no, cioè un'informazione di tipo booleano;

2. si può associare l'informazione di cui al punto precedente direttamente al riferimento all'oggetto interessato, cioè includerla nell'implementazione della connessione stessa.

La definizione della classe `BooleanInstanceRelation` consiste nel dichiarare che si tratta di una specializzazione della classe `Dictionary` predefinita da Objectworks(r)/Smalltalk. Di maggiore interesse è l'implementazione del

```
add: objectId
   "connects a new object"

   self at:  objectId put:  false
```

Figura 12.6. Metodo `add:` della classe `BooleanInstanceRelation`

metodo `add:` riportata nella figura 12.6. Questo metodo è invocato per collegare nuovi oggetti mediante la connessione in modo analogo all'omonimo metodo della classe `InstanceConnection`. L'aggiunta di un oggetto implica l'inserimento di una nuova voce in un'istanza di `Dictionary`, cioè si tratta di introdurre una chiave ed il corrispondente valore. Si utilizza come chiave l'oggetto collegato, cioè il suo identificatore, e come 'value' il valore booleano che indica se la condizione è soddisfatta oppure no.

Le istanze di questa classe sono utilizzate per l'attributo `child` della classe `Task` utilizzato nel modello di processo per individuare le sottoattività. Il collegamento dei figli è fatto nel momento in cui questi sono creati e quindi non sono in stato `#completed`. Poichè la condizione che interessa controllare sulle sottoattività è se siano in stato `#completed` oppure no, il valore che si memorizza al momento della creazione del collegamento è `false`. La nuova voce è creata mandando il messaggio `at:put:` definito per la classe `Dictionary` che richiede come argomenti la chiave ed il valore associato, cioè `false`.

Le istanze di questa classe, come detto in precedenza, sono utilizzate anche per l'attributo `predecessors` di `Task` ed il collegamento dei predecessori è fatto al momento dell'istanziazione dell'attività. Dal momento che si fa uso dell'istanziazione incrementale, gli oggetti `Task` sono creati, e quindi i loro attributi inizializzati, quando tutti i predecessori soddisfano la condizione associata. Ciò significa che nel momento in cui un nuovo predecessore è collegato la sua precondizione è vera e quindi il valore associato alla chiave deve essere `true`. Per questo motivo la classe `BooelanInstanceRelation` mette a disposizine anche il servizio `addTrue:` che funziona in modo analogo ad `add:`, ma utilizza `true` come valore nella nuova voce del dizionario.

Poichè questa classe è stata concepita per l'implementazione di connessioni tra istanze di `Task`, i selettori ed i commenti dei metodi si riferiscono spesso a tale uso, anche se le classi possono essere utilizzate per scopi più

```
allCompletedOfClass: taskClass
   "return True if all Children of TaskClass class are completed"

   self keysAndValuesDo:  [:child :completed | child class ==
taskClass & completed not ifTrue:  [↑false]].
   ↑true
```

Figura 12.7. Metodo `allCompletedOfClass:` della classe `BooleanInstanceRelation`

generali. Ad esempio il metodo riportato nella figura 12.7, controlla che tutte le istanze della classe passata come argomento dell'invocazione collegate dalla connessione destinataria del messaggio, soddisfino la condizione. Per aumentare la leggibilità del codice che del metodo che invia il messaggio, il selettore fa riferimento esplicito al controllo della condizione di completezza, anche se in realtà il metodo non fa altro che controllare i valori memorizzti nel dizionario e può quindi essere utilizzato in casi più generali.

12.4.2 Connessioni di livello classe

L'implementazione delle connessioni di livello classe è fatta secondo gli stessi principi utilizzati per le connessioni di livello istanza ed offrendo metodi del tutto analoghi per la gestione, ma è complicata dal meccanismo stesso in cui Objectworks(r)/Smalltalk tratta la rappresentazione delle classi.

Il concetto che sta alla base dell'implementazione delle connessioni di livello classe, è la memorizzazione del riferimento alla classe connessa in un attributo. Si tratta allora di definire, per le classi che necessitano di una relazione con una o più altre classi, una 'class variable' e di assegnarle il riferimento alla classe voluta, oppure ad un oggetto che implementi la connessione mantenendo i riferimenti alle varie classi.

Il problema nasce quando una classe abbia delle specializzazioni che necessitano della stessa connessione. Quando si dichiara un attributo di livello classe, esso è visibile a tutte le istanze della classe, alle sottoclassi ed ai loro oggetti. Però il valore che esso contiene rimane legato alla classe per cui è dichiarato e, come affermato nell'appendice B.1.2, la 'class variable' non può essere ridichiarata come qualunque altro attributo. Allora quando una sottoclasse accede alla variabile in questione, vi trova il valore assegnato ad

essa dalla superclasse; non può modificarne il contenuto, perché eliminerebbe i dati necessari alla superclasse.

La soluzione che è sembrata più efficiente è stata quella di dichiarare l'attributo nella classe più generale della gerarchia e quindi utilizzare un oggetto di tipo `Dictionary` per implementare la connessione. In questo modo è possibile creare una voce la cui chiave è l'identificatore delle sottoclassi interessate ad utilizzare la connessione.

Classe `ClassRelation`

La più generale connessione di livello classe è implementata mediante le istanze di `ClassRelation` la cui definizione è riportata nella figura 12.8. Essa è una specializzazione della classe `Dictionary`.

```
Dictionary variableSubclass:  #ClassRelation
   instanceVariableNames:  ''
   classVariableNames:  ''
   poolDictionaries:  ''
   category:  'Relations'
```

Figura 12.8. Definizione della classe `ClassRelation`

Ogni classe che entri a far parte di una relazione di livello classe deve inserire una propria voce nel dizionario e ciò è fatto invocando il metodo `new:` la cui implementazione è riportata nella figura 12.9. Mentre l'assegnazione di un oggetto `ClassRelation` alla variabile di classe utilizzata per la connessione è fatta durante l'inizializzazione della classe più generale in cui la variabile è stat dichiarata, l'inserimento della nuova voce viene fatto durante l'inizializzazione delle sottoclassi. A questo proposito si deve mettere in evidenza che tutte le classi che hanno necessità di utilizzare connessioni di livello classe, devono necessariamente disporre di un metodo di inizializzazione che inserisce la voce a loro corrispondente nel dizionario che implementa la connessione. Il metodo di inizializzazione deve essere eseguito almeno una volta da quando la classe è stata creata, a quando si comincia la simulazione del processo implementato.

Dalla figura 12.9 si nota che la classe interessata deve passare come argomento della chiamata il proprio identificatore che viene inserito come chiave nel dizionario che implementa la connessione. Il valore corrispondente è un

```
new: relatingClass
   "create a new entry for relating class"

   self at:  relatingClass put:  Set new
```

Figura 12.9. Metodo `new:` della classe `ClassRelation`

insieme vuoto, cioè un'istanza di `Set`, in cui verranno in seguito memorizzate le classi collegate. Il messaggio utilizzato avrà la forma:

```
Connection new:  self
```

Si può notare come l'attributo che implementa la connessione abbia un nome che comincia con una lettera maiuscola poichè si tratta di una 'class variable'. Inoltre il selettore del metodo è `new:` anche se in realtà non è creata un'istanza, ma il procedimento è analogo all'istanziazione di un oggetto che sarebbe necessaria per l'inizializzazione di un normale attributo.

Il metodo `of:is:` riportato nella figura 12.10 permette di connettere una nuova classe alla classe che lo invoca, ammesso che questa abbia correttamente creato la propria voce con il metodo `new:`. Si vede che la classe da collegare è aggiunta all'insieme individuato mediante la chiave costituita dall'identificatore della classe chiamante che viene passato come argomento `relatingClass` dell'invocazione.

```
of: relatingClass is: relatedClass
"stores the related class"

   (self at:  relatingClass)
      add:  relatedClass
```

Figura 12.10. Metodo `of:is:` della classe `ClassRelation`

Volendo ricavare l'insieme degli oggetti connessi non è più sufficiente l'attributo che implementa la relazione, come nel caso delle connessioni di livello classe, ma è necessario utilizzare un apposito metodo (vedi figura 12.11) che restituisce il valore in corrispondenza della chiave costituita dall'identificatore della classe interessata.

```
of: relatingClass
   "return a set containing related classes"

   ↑self at:  relatingClass ifAbsent:  [Set new]
```

Figura 12.11. Metodo `of:` della classe `ClassRelation`

Classe `ChildrenClassRelation`

Un caso particolare di connessione di livello classe è rappresentato della relazione che lega le istanze di `Task` alle sottoattività in cui possono decomporsi. Infatti, come spiegato nella sezione 10.2, il meccanismo di istanziazione delle sottoattività è influenzato dal fatto che il numero di oggetti da creare sia staticamente determinabile, e quindi noto a priori, oppure dipenda dal processo stesso. Allora è necessario associare ad ogni figlio, se lo si conosce, il numero di istanze che sono necessarie per poi utilizzare quest'informazione secondo quanto descritto nella sezione 13.4. La classe `ChildrenClassRelation` è stata creata appositamente per rispondere a queste esigenze ed è definita come specializzazione di `ClassRelation`.

La differenza fondamentale tra le due classi si può vedere dalla ridefinizione del metodo `new:` riportata nella figura 12.12. La nuova voce del dizionario ha come valore un oggetto `Dictionary`, e non più `Set` come nel caso di `ClassRelation`. Il perché di questo fatto si capisce osservando il metodo per il collegamento della classe di una sottoattività, riportato nella figura 12.13; la nuova classe diviene la chiave di una nuova voce del dizionario relativo alla classe padre (che ha invocato il metodo) il cui valore è il numero di istanze necessarie per quella classe. Tutte queste informazioni sono gli argomenti dell'invocazione che ha la forma:

```
Children of:  self is:  SelectTool numberOfIstances:  1
```

Se il numero di istanze non è noto a priori viene memorizzato il valore zero a cui il metodo di istanziazione delle sottoattività ragirà in modo opportuno.

```
new: fatherClass
   "creates a new entry for the new father"

   self at:  fatherClass put:  Dictionary new
```

Figura 12.12. Metodo `new:` della classe `ChildrenClassRelation`

```
of: fatherClass is: childId numberOfInstances: inst
   "stores the child class and the number of instances wanted for
   that task"

   (self at:  fatherClass)
   at:  childId put:  inst
```

Figura 12.13. Metodo `of:is:numberOfInstances:` della classe `ChildrenClassRelation`

12.5 La classe `DoubleArray`

La categoria `Utilities` contiene un certo numero di classi definite appositamente per l'implementazione di S^3, ma non direttamente legate al processo. Una di queste è già stata introdotta nella sezione 12.3.4, ed è la classe `Controlled`; altre sono descritte in questa sezione.

La classe `DoubleArray` è stata introdotta per fornire al sistema una struttura che può essere vista in due modi differenti:

1. una lista, ogni elemento della quale è una coppia di oggetti;
2. due liste che hanno la stessa dimensione ed i cui elementi nella stessa posizione sono strettamente correlati.

L'implementazione è stata fatta seguendo la struttura suggerita dal secondo punto di vista; come si può notare dalla definizione della classe riportata nella figura 12.14, si hanno due attributi che sono appunto le due liste. Questo fatto si può comprendere facilmente osservando il metodo di inizializzazione delle istanze riportato in figura 12.15 che assegna oggetti della classe `Array` ad entrambe gli attributi.

```
Object subclass:  #DoubleArray
   instanceVariableNames:  'firstElement secondElement '
   classVariableNames:  ''
   poolDictionaries:  ''
   category:  'Utilities'
```

Figura 12.14. Definizione della classe `DoubleArray`

```
initialize
   "initializes a new instance"

   firstElement := Array new.
   secondElement := Array new
```

Figura 12.15. Metodo `initialize` della classe `DoubleArray`

La classe mette quindi a disposizione tutta una serie di metodi per la manipolazione delle sue istanze in modo che possano essere trattate indifferentemente secondo i due punti di vista espressi in precedenza. Questi metodi non hanno alcuna importanza concettuale e pertanto non vengono descritti nel dettaglio, ma a mano a mano che saranno utilizzati se ne illustrerà il valore restituito.

13

Implementazione delle classi del progetto di S^3

Dopo aver trattato i concetti alla base dell'implementazione di S^3, si vuole spiegare nel dettaglio come sono state definite le varie classi presentate nel progetto. La classe `Task` è il nucleo di S^3 perché i suoi metodi realizzano la maggior parte dei meccanismi che caratterizzano l'ambiente ed in particolare la gestione dei dati e degli strumenti, l'interazione con le persone e la coordinazione tra le attività. Il meccanismo di maggior rilievo è quello della decomposizione delle attività mediante l'istanziazione incrementale che è trattato nel dettaglio in questo capitolo. Infine si sono fornite informazioni circa il funzionamento della classe definita dal sistema `AssignTasks`, che è alla base dell'assegnazione gerarchica della persone alle attività.

13.1 Le persone

In questa sezione viene descritta l'implementazione delle classi che sono utilizzate per modellizzare le persone (`Person`) e per gestire il principale tipo di comunicazione tra queste ed il processo (`Agenda`). Sebbene queste classi siano contenute in 'class category' differenti, vengono trattate insieme perché sono strettamente legate tanto da esservi una corrispondenza biunivoca tra esse.

13.1.1 La classe `Person`

Per la modellizzazione delle persone si è introdotta la classe `Person` che è stata definita come mostrato nella figura 13.1. Si può immediatamente notare che questo è il primo caso che incontriamo di utilizzo della classe `Controlled` da cui quella in questione deriva direttamente per ererditarietà. L'utilità del creare `Person` come specializzazione di `Controlled` verrà illustrata trattando l'implementazione dei metodi.

```
Controlled subclass:  #Person
   instanceVariableNames:  'chosenTools agenda name '
   classVariableNames:  ''
   poolDictionaries:  ''
   category:  'Persons'
```

Figura 13.1. Definizione della classe `Person`

L'attributo `chosenTools` è utilizzato per implementare le connessioni con le istanze delle sottoclassi di `Tool` descritta durante la presentazione del progetto e si è utilizzato lo stesso nome usato nel 'design'. Si può notare che questa connessione è mantenuta solamente tra istanze, perché non è utile la corrispondente relazione tra classi. Inoltre la classe `Person` nel progetto prevede anche altre connessioni per cui non compare nessun attributo né di livello istanza, né di livello classe. Questo perché tali connessioni non sono bidirezionali ed è utile solamente che l'altro oggetto conosca l'istanza di `Person` a cui è collegato e non viceversa.

L'attributo `agenda` permette di identificare la principale interfaccia per la comunicazione con la persona modellizzata che è descritta in dettaglio nella sezione 13.1.2. La variabile di istanza `name` serve esclusivamente per memorizzare il nome della persona modellizzata; in un sistema realmente utilizzato sarebbe necessario definire altre variabili per mentenere varie informazioni riguardo ad ogni individuo coinvolto nel processo ed al suo tipo di esperienza e preparazione.

```
new: personName
   "creates a new instance and stores person name"

   | newPerson |
   newPerson := super new.
   ↑newPerson initialize:  personName
```

Figura 13.2. Metodo `new:` della classe `Person`

Creazione delle istanze

La creazione delle istanze avviene mediante l'invocazione del metodo `new:` della classe `Person` (figura 13.2) cui viene fornito il nome della persona modellizzata dalla nuova istanza. Questo metodo non fa altro che creare la nuova istanza mandando il messaggio `new` alla propria superclasse e chiamando il metodo di inizializzazione sulla nuova istanza.

```
initialize: personName
   "stores person's name"

   name := personName.
   agenda := Agenda ofPersonNamed:  name.
   chosenTools := InstanceConnection new
```

Figura 13.3. Metodo `initialize:` della classe `Person`

Inizializzazione delle istanze

L'implementazione del metodo `initialize:` è riportata nella figura 13.3 e realizza l'inizializzazione degli attributi della nuova istanza. Alla variabile di istanza `name` è semplicemente assegnata la stringa passata come argomento dell'invocazione che contiene il nome della persona modellizzata. Per implementare una connessione, alla 'instance variable' `chosenTools` è assegnata un'istanza della classe `InstanceConnection`. Infine viene creato un oggetto che realizza l'interfaccia di comunicazione dell'utente invocando il metodo `of:named:` della classe `Agenda` che è descritto nella sezione 13.1.2.

Esempio di uso di una connessione in modo protetto

Nella figura 13.4 è riportato il codice del metodo `choiceIs:` che viene utilizzato per memorizzare uno degli strumenti automatici scelti dalla persona associata all'oggetto `Person` su cui è invocato. Esso è un tipico esempio dell'utilizzo di una connessione e di accesso controllato all'attributo che la implementa.

```
choiceIs: tool
   "record chosen tool of class toolClass"

   | tmp |
   self critical:  [tmp := chosenTools add:  tool].
   ↑tmp
```

Figura 13.4. Metodo `choiceIs:` della classe `Person`

Quando una persona esprime la propria preferenza per un certo strumento automatico, questo deve essere connesso all'oggetto che la rappresenta dalla connessione `ChosenTools/ChosenBy` di cui si parla nella sezione 9.4.1. Tale legame viene imposto invocando il metodo `choiceIs:` e fornendo come argomento l'identificatore dell'istanza di `Tool` che rappresenta lo strumento in questione. Il collegamento vero e proprio avviene inviando alla connessione identificata da `chosenTools` il messaggio `add:`. Però dal momento che i metodi degli oggetti della classe `Person` possono essere invocati da più di un processo concorrente, è necessario utilizzare il supporto per la protezione ereditato da `Controlled`. Per questo il messaggio all'oggetto identificato da `chosenTools` è racchiuso in un blocco e non è eseguito direttamente. Viene invocato il metodo `critical:` per l'istanza di `Person` in questione garantendo la mutua esclusione sull'esecuzione di `add:` per la connessione interessata. La variabile temporanea `tmp` è utilizzata per ritornare il valore restituito dall'invocazione di `add:` solo dopo essere usciti dal blocco che costituisce la regione critica. In realtà, per come è implementato il metodo `critical:` da Objectworks(r)/Smalltalk, si poterebbe anche usare il ritorno da metodo (↑) all'interno del blocco, ma questa implementazione è più pulita.

13.1.2 La classe `Agenda`

Come accennato più volte il sistema necessita di un certo numero di interfacce mediante cui può interagire con gli utenti; quella principale è detta *agenda* in quanto riporta, per ogni persona coinvolta nel processo, l'elenco delle azioni per cui è richiesto il suo intevento. Questa interfaccia appare come una lista in cui ogni elemento identifica un'azione per cui è richiesto l'intervento del corrispondente utente; per ogni voce è possibile visualizzare una descrizione più dettagliata tramite un apposito comando del 'menu' associato alla finestra. La figura 13.5 mostra un'Agenda per cui è stata visualizzata una di queste descrizioni.

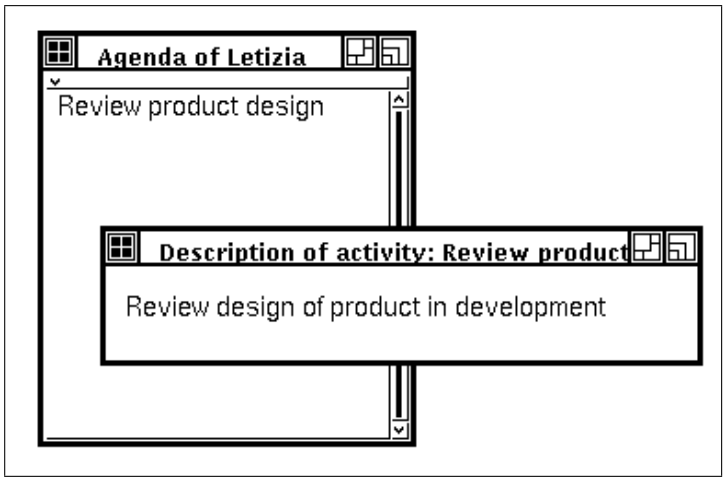

Figura 13.5. Aspetto dell'interfaccia `Agenda`

Definizione

Questa interfaccia è gestita dalla classe `Agenda` la cui definizione è riportata nella figura 13.6. Per l'implementazione delle interfacce grafiche si è utilizzata la struttura 'Model-View-Controller' descritta nell'appendice B.7.1 e la classe `Agenda` costituisce proprio il componente 'model'; per questo motivo essa è dichiarata come specializzazione della classe `Model` definita da Objectworks(r)/Smalltalk per fornire le funzionalità e caratteristiche tipiche dell'omonimo componente.

La variabile di istanza `list` viene utilizzata per memorizzare la lista delle azioni visualizzata dall'interfaccia. La variabile `semaphore` è inizializzata

```
Model subclass:  #Agenda
   instanceVariableNames:  'selection list semaphore '
   classVariableNames:  ''
   poolDictionaries:  ''
   category:  'User interfaces'
```

Figura 13.6. Definizione della classe `Agenda`

come semaforo per la mutua esclusione ed serve per controllare gli accessi a `list`. Infatti questo attributo può essere acceduto contemporaneamente

1. dal processo che controlla l'interfaccia, per aggiornare le informazioni visualizzate;

2. dal flusso di esecuzione proveniente da un'istanza di una sottoclasse di `Task` che vuole aggiungere una nuova azione alla lista.

In questo caso si tratta del primo tipo di protezione proposto nella sezione 12.3.2 in quanto l'attributo da proteggere è solamente uno. Diversamente si sarebbero potuti utilizzare i meccanismi offerti dalla classe `Controlled`, ma ciò sarebbe stato complicato dal fatto che `Agenda` deve essere specializzazione di `Model`.

La variabile di istanza `selection` permette di memorizzare l'elemento correntemente selezionato nella lista. Quando l'utente seleziona un elemento della lista il controllore informa il 'model' associato (in questo caso l'istanza di `Agenda`) invocandone un metodo che ha come argomento il numero di tale elemento; tale numero deve essere memorizzato dal 'model' e per questo scopo viene utilizzato `selection`.

Inizializzazione delle istanze

Il metodo di creazione delle istanze invocato nella figura 13.3 invia il messaggio `new` alla superclasse ed invoca il metodo di inizializzazione riportato nella figura 13.7. Questo pone a zero l'attributo contenente la selezione nella lista ed inizializza il semaforo per proteggere quet'ultima in modo da utilizzarlo per la mutua esclusione come descritto nell'appendice B.8.3. Alla variabile di istanza `list` è assegnato un oggetto della classe di utilità `DoubleArray` descritta nella sezione 12.5; l'utilità di questa assegnazione è chiara parlando della gestione della lista associata all'interfaccia.

```
personNameIs: personName
   "initialize the Agenda"

   list := DoubleArray new.
   semaphore := Semaphore forMutualExclusion.
   selection := 0.
   self displayWithTitle:  'Agenda of ' , personName
```

Figura 13.7. Metodo `personNameIs:` della classe `Agenda`

Il messaggio `displayWithTitle:`, riportato nella figura 13.8, viene inviato direttamente dal metodo di inizializzazione e serve a visualizzare la finestra contenente l'interfaccia. A tale finestra è assegnato come titolo l'argomento dell'invocazione che è una stringa contenente il nome della persona a cui è associata l'interfaccia in questione.

```
displayWithTitle: title
   "creates the window"

   | agendaView container agendaWindow |
   container := CompositePart new.
   agendaWindow := ScheduledWindow new.
   agendaWindow label:  title.
   agendaView := SelectionInListView
         noDelimitersOn:  self
         aspect:  #agenda
         change:  #selectionIs:
         list:  #activityList
         menu:  #activityListMenu
         initialSelection:  nil.
   container add:  (LookPreferences edgeDecorator on:  agendaView)
      borderedIn:  (0 @ 0 extent:  1 @ 1).
   agendaWindow component:  container.
   agendaWindow controller:  NoCloseController new.
   agendaWindow openNoTerminateIn:  (Window currentOrigin + 50
extent:  200 @ 200)
```

Figura 13.8. Metodo `displayWithTitle:` della classe `Agenda`

La creazione della finestra e della 'view' sono fatte secondo le modalità

descritte nell'appendice B.7; in particolare si può vedere un esempio dell'invio del messaggio per l'istanziazione della 'view' per operare selezioni in una lista. Il modello collegato ad essa è `self`, cioè l'oggetto che implementa le interazioni dell'agenda con il sistema. I metodi `selectionIs:`, `activityList` ed `activityListMenu` sono definiti per la classe `Agenda`; l'argomento che segue la parola chiave `initialSelection:` è `nil` perché al momento della prima visualizzazione non si deve avere alcun elemento selezionato.

Gestione della lista associata

Come accennato in precedenza l'agenda è utilizzata dalle varie attività del processo quando necessitano di interagire con l'utente per svolgere alcuni compiti. In tale circostanza viene inviato il messaggio `recordActivity:-from:description:` all'istanza di `Person` che modellizza il responsabile dell'attività in questione; sebbene per motivi di notazione si sia utilizzato un diverso selettore, questo messaggio corrisponde a `GetUserAttention` descritto nella presentazione del progetto del modello di processo. L'esecuzione del metodo in questione, il cui codice è riportato nella figura 13.9, invia un messaggio con gli stessi argomenti all'oggetto `Agenda` associato. Questa invocazione è il primo esempio di creazione di un nuovo processo per l'esecuzione del metodo invocato. Infatti l'oggetto `Person` non aspetta alcuna risposta dall'istanza di `Agenda` alla cui lista chiede di aggiungere un nuovo elemento. Sarà l'interfaccia stessa che, a seguito delle azioni dell'utente, provvederà a fornire le opportune informazioni all'attività interessata che è identificata dall'argomento `taskId`.

```
recordActivity: activityName from: taskId description: descr
   "request the Agenda to record a new activity received from Task
   taskId for aMatter"

   [agenda
      record:  activityName
      from:  taskId
      description:  descr] fork
```

Figura 13.9. Metodo `recordActivity:from:description:` della classe `Person`

```
record: elementName from: taskId description: descr
   "records a new activity received from Task taskId for aMatter"

   | temp |
   temp := Array with:  descr with:  taskId.
   semaphore critical:  [list add:  elementName with:  temp].
   self changed:  #agenda
```

Figura 13.10. Metodo `record:from:description:` della classe `Agenda`

Il metodo `record:from:description:` è riportato nella figura 13.10 e non fa altro che inserire un nuovo elemento nella lista associata all'interfaccia. La variabile di istanza `list` non contiene solamente le voci che sono mostrate nella lista di selezione che compare nella finestra, ma anche delle informazioni per la gestione dell'interfaccia. In primo luogo per ogni elemento della lista occorre memorizzare la descrizione correlata che può essere visualizzata mediante la selezione dell'apposito comando del 'menu' associato alla 'view'. Inoltre, poiché ogni elemento della lista corrisponde ad una richiesta di interazione da parte di un'attività, è tenuta traccia anche dell'istanza di `Task` interessata che è identificata dall'argomento `taskId`. Allora `list` viene organizzata come una lista di coppie (istanza di `DoubleArray`) ognuna delle quali è costituita dalla voce che compare sul video e da una seconda coppia; quest'ultima è formata dalla descrizione della voce associata e dall'identificatore dell'attività interessata. Si può notare che l'accesso a `list` è fatto in modo controllato utilizzando la regione critica associata al semaforo `semaphore`.

Notifica alle attività

Dopo che l'utente ha selezionato uno degli elementi della lista, può decidere di eseguire l'azione associata utilizzando l'apposito comando `Execute` del 'menu'. Questo corrisponde all'invocazione del metodo `gotAttention` del 'model' associato (istanza di `Agenda`) la cui implementazione è riportata nella figura 13.11.

Per ricavare l'attività interessata si recupera il secondo oggetto che costituisce ogni coppia della lista `list`, mediante il metodo `secondAt:` della classe `DoubleArray`; su questo oggetto, che è un vettore di due elementi, si

```
gotAttention
   "notify task that user is ready"

   | receiver |
   semaphore
      critical:
         [receiver := (list secondAt:  selection)
               at:  2.
         list removePosition:  selection].
   receiver gotUserAttention.
   self changed:  #agenda
```

Figura 13.11. Metodo `gotAttention` della classe `Agenda`

invoca il metodo `at:` della classe `Array` per ottenere la seconda componente che è proprio l'attività interessata.

A questo punto la voce in questione viene eliminata da `list` invocando il metodo `removePosition`; per fare in modo che la nuova lista così ottenuta sia effettivamente visualizzata, viene invocato il metodo `changed:` sull'istanza di `Agenda` che causa l'aggiornamento della 'view' associata grazie al meccanismo delle dipendenze di Smalltalk.

La notifica vera e pripria all'attività avviene mediante l'invio del messaggio `gotUserAttention` la cui implementazione è riportata nella figura 13.29 trattando della classe `Task`. Nella sezione 10.5.3 si è detto che l'interfaccia comunica con l'oggetto che modellizza il responsabile e questo a sua volta informa l'attività interessata. Questo è certamente più corretto dal punto di vista formale, dal momento che questa interfaccia è associata ad una persona ed è quindi connessa ad una ben precisa istanza di `Person`. Però, per ragioni di efficienza (e dal momento che Smalltalk non supporta modularità per cui l'interfaccia può accedere direttamente alle istanze di `Task`), nell'implementazione l'interfaccia invia direttamente il messaggio all'istanza di `Task` interessata.

13.2 La categoria `Data`

Nella 'class category' `Data` sono contenute le classi che nel progetto costituiscono il soggetto `Data`. La radice della struttura 'gen/spec' è la classe

`Data` che implementa molti dei meccanismi propri del sottomodello dei dati. A differenza della maggior parte delle altre gerarchie di specializzazione, sia quelle predefinite da Objectworks(r)/Smalltalk, che quelle create appositamente per la realizzazione di S^3 (ad esempio `Task` e le sue sottoclassi), i metodi implementati nella classe più generale `Data` non possono essere utilizzati tali e quali nelle sottoclassi, ma devono essere ridefiniti in esse. Questo fatto verrà chiarito maggiormente nella sezione 13.2.1

13.2.1 La classe `Data`

Nella figura 13.12 è riportata la definizione della classe `Data` in cui si nota che le connessioni che la riguardano sono in numero minore rispetto a quelle introdotte nel progetto. Questo è dovuto al fatto che nella maggior parte dei casi è sufficiente che l'altro oggetto connesso conosca quello della classe `Data`, ma non è necessario viceversa.

```
Controlled subclass:  #Data
   instanceVariableNames:  'productors faultsDocuments '
   classVariableNames:  'Productors '
   poolDictionaries:  ''
   category:  'Data'
```

Figura 13.12. Definizione della classe `Data`

Metodi per la gestione delle connessioni

I metodi definiti nella classe `Data` sono stati inclusi quasi tutti nel protocollo `relation handling` e sono utilizzati per navigare da un oggetto all'altro attraverso le connessioni tra essi. Un certo numero di questi metodi è utilizzato per creare le connessioni, mentre gli altri servono per restituire la classe o la siangola istanza connessa.

Come evidenziato dalla definizione della classe riportata nella figura 13.12, le connessioni definite per `Data` sono solamente quellea verso `Task` (`Output/Productors`) e quella verso `FeedbackDocument`. Di maggior rilievo sono quelle definite tra le sottoclassi di `Data` che permettono di stabilire la configurazione del prodotto e di individuare due tipi di relazioni tra i dati che sono illustrate nella sezione seguente.

13.2.2 La classe `SourceFile` e le sue connessioni

Per esemplificare le connessioni tra le sottoclassi di `Data` consideriamo la classe `SourceFile` definita secondo lo schema riportato nella figura 13.13. Le variabili di istanza che compaiono nella definizione sono tutte utilizzate per creare connessioni verso altre sottoclassi di `Data`. Come si vede nella figura 9.9 che riporta un modello di dati predefinito contenuto nel soggetto `Data`, `SourceFile` è connessa a `DesignDocument` (`DescribedBy`), a `commentFile` (`CommentedBy`) ed a `CompilableSourceFile` (`Used`). Oltre a queste connessioni, tra gli oggetti `Data` esistono due tipi di relazioni che vengono utilizzati per correlare le attività che sui dati agiscono. Queste relazioni non sono mantenute mediante connessioni tra gli oggetti interessati, ma sono ricavate, mediante l'invocazione di appositi metodi, dalle connessioni che stabiliscono la struttura del prodotto. Nel seguito è riportata una decsrizione di questi due tipi di relazioni e l'implementazione dei due metodi che permettono di utilizzarle per la classe `SourceFile`. L'uso di questi metodi è esemplificato nella sezione 13.4.2.

```
ConfigurationComponent subclass:  #SourceFile
   instanceVariableNames:  'describedBy use commentedBy '
   classVariableNames:  'DescribedBy Use '
   poolDictionaries:  ''
   category:  'Data'
```

Figura 13.13. Definizione della classe `SourceFile`

Dati condizionanti

Sono tutti gli oggetti di `Data` che condizionano in qualche modo, direttamente o indirettamente, la produzione dell'istanza in questione. Questa relazione non è specificata mediante una connessione tra istanze, ma semplicemente ogni sottoclasse di `Data` ridefinisce il metodo `conditioningDataFrom:` che ritorna un insieme contentete tutti gli oggetti che condizionano quello su cui il metodo è stato invocato. La definizione di questo metodo per la classe `SourceFile` è riportata nella figura 13.14 e può essere utile fare alcune considerazioni su di essa.

L'istanza di `Set` da ritornare viene creata come variabile temporanea ed

```
conditioningDataFrom: sender
   "return a Set containing all Data objects that somehow
condition self."
   | tmp temp |
   tmp := Set new.
   self critical:  [temp := describedBy].
   temp do:  [:obj | obj = sender
         ifFalse:
            [tmp addAll:  (obj conditioningDataFrom:  self).
            tmp add:  obj]].
   self critical:  [temp := commentedBy].
   temp notNil & (temp ~= sender)
   ifTrue:
         [tmp addAll:  (temp conditioningDataFrom:  self).
         tmp add:  temp].
   ↑tmp
```

Figura 13.14. Metodo `conditioningDataFrom:` di `Data`

in essa sono inclusi gli oggetti connessi da `describedBy` e `commentedBy`. C'è da segnalare il fatto che l'omonimo metodo è invocato anche sulle istanze trovate, se non coincidono con l'oggetto che ha invocato il metodo in esecuzione il cui identificatore è contenuto nell'argomento `sender` (per evitare infinite invocazioni annidate); l'insieme ottenuto è incluso in quello temporaneo che è restituito al termine.

Dati correlati

Si tratta di tutti gli oggetti `Data` che sono in qualche modo, anche indirettamente, correlati all'istanza presa in esame. Tra questi non sono però inclusi i dati per la cui produzione è stato utilizzato l'oggetto in questione. Anche questa relazione è implementata ridefinendo il metodo `relatedDataFrom:` che ritorna un insieme contentete tutti gli oggetti correlati a quello su cui il metodo è stato invocato. Quali siano questi oggetti viene ricavato in base alle connessioni a cui partecipa l'oggetto in questione. Nella figura 13.15 è riportata l'implementazione del metodo `relatedDataFrom:` per la classe `SourceFile`. Si può notare che gli oggetti correlati alle istanze di questa classe sono un soprainsieme di quelli che le condizionano; infatti l'insieme da restituire è creato associandogli il valore di ritorno dell'invocazione del

metodo `conditioningDataFrom:` sull'oggetto `self`.

```
relatedDataFrom: sender
   "return a Set containing all Data objects that somehow are
related with self."

   | tmp temp |
   tmp := self conditioningDataFrom:  sender.
   self critical:  [temp := use].
   temp do:  [:obj | obj = sender ifFalse:  [tmp add:  obj]].
   self critical:  [temp := faultsDocuments].
   tmp addAll:  temp.
   ↑tmp
```

Figura 13.15. Metodo `relatedDataFrom:` di `SourceFile`

Attività condizionanti

Come accennato in precedenza le informazioni derivanti dalle due relazioni suddette vengono utilizzate per ricavare relazioni tra attività. Allora la classe `Data` fornisce il metodo `conditioningTasks` che restituisce un insieme contenente tutte le istanze di `Task` che condizionano l'attività che produce l'oggetto `Data` sui cui il metodo è invocato. Infatti, come si vede dal codice riportato nella figura 13.16, l'insieme da restituire è ottenuto raccogliendo tutte le istanze di `Task` connesse come produttori dei dati condizionanti l'oggetto `Data` che sta eseguendo il metodo.

`conditioningTasks` invoca su `self` il metodo `conditioningDataFrom:` che deve essere implementato nelle sottoclassi; quindi `conditioningTasks`, sebbene sia definito in `Data`, può essere eseguito solamente sugli oggetti delle sottoclassi.

13.3 La classe `Task`

È senza dubbio la classe centrale nella realizzazione di S^3 in quanto nella sua implementazione è fornito supporto alla maggior parte dei meccanismi su cui si basa il funzionamento dell'intero ambiente, in particolare per quanto riguarda la comunicazione e la coordinazione delle attività. Nella figura 13.17

conditioningTasks

```
   "return a Set containing all tasks that produce as output data
that somehow condition self."

   | collection |
   collection := Set new.
   (self conditioningDataFrom:  self)
   do:
   [:obj |
   | tmp |
   (tmp := obj productors) size isZero ifTrue:  [↑nil].
   collection addAll:  tmp].
   ↑collection
```

Figura 13.16. Metodo `conditioningTasks` di `Data`

è riportata la definizione della classe `Task` da cui si può in primo luogo notare che si tratta di una specializzazione di `Controlled` in quanto le sue istanze sono oggetti attivi, secondo quanto spiegato nella sezione 12.3.1.

```
Controlled subclass:  #Task
   instanceVariableNames:  'state toBeRestarted associatedTools
father children assignedRoles responsible successors predecessors
waitingFeedback input output userAction '
   classVariableNames:  'Children Input Output Predecessors
ResponsibleRole WaitingFeedback '
   poolDictionaries:  ''
   category:  'Task'
```

Figura 13.17. Definizione della classe `Task`

Quasi tutti gli attributi della classe `Task` sono utilizzati per implementare le connessioni (fanno eccezione `state`, `toBeRestarted` e `userAction`) e il loro impiego verrà illustrato trattando dell'implementazione dei vari metodi. Vale la pena comunque di evidenziare il fatto che alcuni dei nomi delle 'instance variable' appaiono anche come 'class variable', semplicemente avendo la prima lettera del nome maiuscola, come richiedono le convenzioni adottate in Smalltalk. Questo fatto indica che le connessioni implementate da

questi attributi sono necessarie sia a livello classe che a livello istanza, ma ciò non vale per qualsiasi connessione come si è ipotizzato per il progetto (sezione 6.5). Inoltre si può notare che la connessione `father/children` a livello delle istanze è bidirezionale, ma a livello classe non si ha un attributo `Father`; infatti, mentre è importante che un'istanza conosca l'oggetto che la ha istanziata per mantenerlo informato circa il proprio stato, non è di nessuna utilità che una sottoclasse di `Task` conosca i tipi di attività dalla cui decomposizione possono provenire i suoi oggetti.

13.3.1 Livello classe

Questo è il primo caso in cui si abbiano attributi e metodi di livello classe (a parte quello per la creazione delle istanze). Le variabili di classe di `Task` sono utilizzate per implementare le connessioni tra classi che esprimono aspetti legati al modello del processo, non tanto alla sua esecuzione. Infatti tutti gli aspetti propri del modello quali la gerarchia di decomposizione delle attività o le connessioni che individuano i tipi di dato prodotti da un'attività, sono fornite e gestite a livello classe. In ogni caso le connessioni di livello classe sono vastamente utilizzate durante l'esecuzione del processo ed i metodi di livello classe sono appunto utilizzati per accedere ad esse.

Inizializzazione delle variabili

I metodi di inizializzazione di livello classe devono essere invocati almeno una volta prima dell'istanziazione di un processo e quindi non è più necessario utilizzarli dal momento che le classi, e quindi le loro variabili e le connessioni stabilite tra esse, hanno persistenza in Objectworks(r)/Smalltalk.

Il metodo di livello classe `initialize` riportato nella figura 13.18, assegna gli opportuni oggetti alle 'class variable' e quindi invoca l'omonimo metodo su ognuna delle sottoclassi.

La classe `Task` non ha istanze proprie e neppure connessioni di livello classe che la interessano, ma deve comunque inizializzare le varibili di classe che devono essere gestite secondo i principi illustrati nella sezione 12.4.2. Il metodo `initialize` invocato sulle sottoclassi provvederà a creare un'opportuna 'entry' nel dizionario costituito dagli oggetti `ClassRelation` da associare alla sottoclasse in questione. Un esempio di tale metodo è quello della classe `DevelopProgram` che è stato riportato nella figura 13.19.

```
initialize
   "initializes class variables"

   Children := ChildrenClassRelation new.
   Predecessors := ClassRelation new.
   WaitingFeedback := ClassRelation new.
   ResponsibleRole := SingleClassRelation new.
   Input := ClassRelation new.
   Output := ClassRelation new.
   self allSubclasses do:  [:subClass | subClass initialize]
```

Figura 13.18. Metodo di livello classe `initialize` di `Task`

Dall'implementazione di questo metodo si può vedere l'utilizzo delle classi `ClassRelation` e relative sottoclassi. Ad esempio la 'class variable' `Children` è un'istanza di `ChildrenClassRelation` ed in primo luogo viene invocato il metodo `new:` per dichiarare che la classe mittente (`DevelopProgram` in questo caso) necessita di una connessione di questo tipo. Quindi viene inviato il messaggio `of:is:numberOfInstances:` per ognuna delle classi che rappresentano un'attività in cui le istanze di `DevelopProgram` possono decomporsi. Si può notare che per `SelectTool`,`Design`, `ReviewDesign` e `Program` tale numero di istanze ,è specificato ed è pari ad uno, come si può ricavare dalle cardinalità delle connessioni presenti nella parte di progetto riportata nella figura 11.3. Per `AssignTasks` il numero di istanze specificato è zero, infatti il fatto che siano create sottoattività corrispondenti ad oggetti di questa classe deve essere determinato durante l'attuazione del processo stesso.

La variabile `Output` è invece un'istanza della classe `ClassRelation`, ma il protocollo utilizzato per stabilire la connessione con la sottoclasse di `Data` che interessa, è praticamente identico. Dal messaggio utilizzato per la creazione della connessione si può immediatamente notare che `Output/Productor` è stata implementata come bidirezionale, infatti `Output` è fatto puntare alla classe `Configuration`, ma nel contempo viene invocato il metodo `addProductor:` di quest'ultima per fare in modo che `DevelopProgram` sia connesso a `Configuration` come uno dei produttori.

Gestione delle connessioni di livello classe

`Task` contiene la dichiarazione di tutta una serie di metodi di livello classe

```
initialize
   "defines children classes"

   Children new:  self.
   Children
      of:  self
      is:  AssignTasks
      numberOfInstances:  0.
   Children
      of:  self
      is:  SelectTool
      numberOfInstances:  1.
   Children
      of:  self
      is:  Design
      numberOfInstances:  1.
   Children
      of:  self
      is:  ReviewDesign
      numberOfInstances:  1.
   Children
      of:  self
      is:  Program
      numberOfInstances:  1.
   ResponsibleRole of:  self is:  ProjectManager.
   Output new:  self.
   Output of:  self is:  (Configuration addProductor:  self).
   Input new:  self.
   Input of:  self is:  (RequirementDocument addProductor:  self)
```

Figura 13.19. Metodo di livello classe `initialize` di `DevelopProgram`

che sono utilizzati principalmente per la gestione delle connessioni. Vediamo come esempio il metodo `neededRoles` riportato nella figura 13.20. Il meccanismo utilizzato per l'assegnazione dell persone alle attività prevede di assegnare alle sottoattività solamente le persone che ricoprono un ruolo richiesto per uno dei responsabili di queste attività o delle loro sottoattività. Allora, per ogni attività cui si devono fare assegnazioni è utile conoscere i ruoli richiesti per il suo responsabile e per i responsabili di eventuali sottoattività. Queste informazioni possono essere ottenute indagando le connessioni

di livello classe `Father/Children` e `ResponsibleRole/CompetenceTasks` e ciò è fatto dal metodo `neededRoles`. Una caratteristica di rilievo di questo metodo è che esso invoca il metodo omonimo su altri oggetti percorrendo così una parte della gerarchia delle attività.

```
neededRoles
   "provide his ResponsibleRole and his children's ResponsibleRoles"

   | roles |
   roles := Set new.
   roles add:  self responsibleRole.
   Children everyChildrenOf:  self do:  [:childClass | childClass
~= self ifTrue:  [childClass neededRoles do:  [:neededRole |
roles add:  neededRole]]].
   ↑roles
```

Figura 13.20. Metodo di livello classe `neededRoles` di `Task`

Il valore restituito è un'istanza di `Set` contenente tutti i ruoli necessari alla classe che sta eseguendo il metodo ed alle classi che modellizzano le sue sottoattività. L'istanza di `Set` è creata con la prima istruzione ed è memorizzata nella variabile temporanea `roles`. Quindi viene inserito in essa il riferimento alla classe che rappresenta il ruolo richiesto per il responsabile della classe che sta eseguendo il metodo. A questo punto si passa a considerare tutte le classi che modellizzano sottoattività mediante l'invocazione del metodo `everyChildrenOf:do:` sulla relazione di classe individuata da `Children`; esso riceve come secondo argomento un blocco che viene eseguito per ogni classe connessa dalla connessione in questione. L'identificatore di questa classe è assegnato all'argomento del blocco `childClass`; in primo luogo viene verificato che la classe in questione non sia quella che sta eseguendo il metodo `neededRoles` (`self`), perché in tal caso si causerebbe un annidamento infinito di chiamate. Se dunque non si verifica questa situazione, si invia il messaggio `neededRoles` alla classe individuata da `childClass` che restituisce un insieme di ruoli. Questi devono ora essere inclusi in `roles` e ciò è fatto sfruttando il metodo `do:` definito per le istanze di `Set` che riceve come argomento un blocco che viene eseguito per ognuno degli elementi dell'insieme.

13.3.2 Livello istanza

La creazione delle istanze della classe `Task` avviene invocando il metodo `new:` a cui è passato come parametro l'identificatore del padre della nuova attività. Questo metodo non fa altro che istanziare un oggetto del tipo voluto e quindi inviargli il messaggio `initialize:` la cui implementazione è riportata nella figura 13.21.

```
initialize: parent
   "initialize all instance variables"

   children := BooleanInstanceRelation new.
   predecessors := BooleanInstanceRelation new.
   successors := InstanceConnection new.
   waitingFeedback := InstanceConnection new.
   input := InstanceConnection new.
   output := InstanceConnection new.
   father := parent.
   toBeRestarted := false.
   userAction := Semaphore new.
```

Figura 13.21. Metodo `initialize:` di `Task`

Il metodo `initialize:` provvede all'inizializzazione degli attributi che permettono di implementare le connessioni di livello istanza necessarie al nuovo oggettto ed assegnare un valore a `toBeRestarted` e `userAction` il cui impiego è illustrato nel seguito. Inoltre l'attività viene connessa al proprio padre assegnando l'argomento dell'invocazione `parent` all'attributo `father`.

13.3.3 Lo stato degli oggetti `Task`

L'unico attributo che non viene inizializzato dal metodo `initialize:` è `state`; esso è utilizzato per memorizzare lo stato dell'attività. Come si è detto illustrando il progetto, un'attività, sebbene istanziata, non è pronta ad iniziare il suo ciclo di vita fino a che non le sono associate le persone che le sono necessarie. Questo compito è svolto in parte dal meccanismo di decomposizione in sottoattività (sezione 13.4.7) ed in parte dalle attività della classe `AssignTasks` (sezione 13.5).

```
endOfAssignation
   "states that all assignations are done"

   self critical:  [self changeState:  #waitingPreconditions]
```

Figura 13.22. Metodo `endOfAssignation` di `Task`

Quando l'assegnazione è terminata viene invocato il metodo `endOfAssignation` (figura 13.22) che provvede a dare il valore `#waitingPreconditions` all'attributo `state`. Questo non è fatto direttamente, ma mediante l'invocazione del metodo `changeState:`.

Il metodo `changeState:`

Il metodo `changeState:` è utilizzato per gestire i cambiamenti di stato delle attività e si fa carico di tutti i controlli che li riguardano. Esso prevede un argomento che può essere di due tipi.

- Il nuovo stato in cui l'oggetto deve transitare; non è detto che il cambiamento di stato avvenga, ma il metodo si fa carico di controllare se sono verificate le opportune condizioni e, se ciò avviene, causa la transizione.
- `nil` quando l'invocazione è fatta per controllare se è possibile una transizione, senza specificare quale. Il metodo si fa carico di ricavare, in base allo stato del processo, quale sia il nuovo stato in cui eventualmente transitare.

L'implementazione di `changeState:` è riportata nella figura 13.23. Nel corpo del metodo si possono idividuare due parti principali.

- La prima parte gestisce tutte quelle chiamate che richiedono la transizione in uno specifico stato espresso dall'argomento `newState`, controllando che tale transizione sia possibile. Se ciò è vero viene aggiornato il valore di `state` rendendolo uguale a `newState`.
- La seconda parte controlla, a seconda dello stato attuale rappresentato dal valore di `state`, se sia possibile transitare in un altro stato. Si può notare che questa parte, essendo eseguita dopo la prima, può far sì che, dopo essere entrati in uno stato a seguito della richiesta esplicita di

```
changeState: newState
   newState = #waitingPreconditions & (state = #completed) |
(newState = #completed)
      ifTrue:
         [state := newState.
         successors do:  [:taskId | [taskId handleTransition:
self] fork].
         [father handleTransition:  self] fork]
      ifFalse:
         [newState = #executing
            ifTrue:
               [state := newState.
               self startExecution.
               ↑self].
         toBeRestarted
            ifTrue:
               [toBeRestarted := false.
               state := #waitingPreconditions]
            ifFalse:  [newState notNil ifTrue:  [state :=
newState]]].
   #waitingPreconditions = state
      ifTrue:
         [predecessors allCompleted ifFalse:  [↑self].
         self changeState:  #executing.
         ↑self].
   #handlingChildren = state ifTrue:  [predecessors allCompleted
         ifTrue:
            [Children childrenAndNumberOf:  self class do:
[:childClass :num | num > 0 ifTrue:  [(children
instancesNumberOfClass:  childClass) isZero ifTrue:  [↑self]]].
            children allCompleted
               ifTrue:
                  [self changeState:  #completed.
                  ↑self]]
         ifFalse:  [self changeState:  #waitingPreconditions]].
   #completed = state & (predecessors allCompleted & children
allCompleted) not ifTrue:  [self changeState:
#waitingPreconditions]
```

Figura 13.23. Metodo `changeState:` di `Task`

transitarvi, si passi direttamente ad un altro perché sono soddisfatte le

condizioni per la nuova transizione.

Il metodo `handleTransition:`

Dall'implementazione di `changeState:` si può notare che, ogni volta che l'attività entra o esce dallo stato `#completed`, viene inviato il messaggio `handleTransition:` su tutti i succesori e sul padre passando come argomento `self`. Si può notare che l'esecuzione del metodo per ogni oggetto interessato è assegnata ad un processo concorrente perché la notifica di terminazione non richiede alcuna risposta, ma soprattutto può innescare l'esecuzione di altre attività.

handleTransition: sender

```
   "declares the task completed"

   self
      critical:
         [(children includesKey:  sender)
            ifTrue:  [children at:  sender put:  (children at:
sender) not]
            ifFalse:  [predecessors at:  sender put:  (predecessors
at:  sender) not].
         self instantiateChildren.
         self changeState:  nil]
```

Figura 13.24. Metodo `handleTransition` di `Task`

Nella figura 13.24 è riportato il codice del metodo `handleTransition` il quale in primo luogo controlla se l'oggetto mittente è un figlio. A seconda dell'esito di tale controllo, inverte il valore booleano associato alla connessione che lega l'oggetto che sta eseguendo il metodo a `sender`. Questa inversione è sufficiente a memorizzare l'avvenuto cambiamento di stato perché, come detto parlando del progetto e ribadito poco sopra, `handleTransition` viene invocato ogni qualvolta una transizione coinvolge lo stato `#completed`. Dunque se la condizione è vera, cioè l'oggetto è nello stato `#completed`, e si ha un'invocazione, questa è dovuta all'uscita da tale stato e quindi la condizione deve divenire falsa. Se invece la condizione è falsa, cioè l'oggetto non è nello stato `#completed`, l'invocazione non può essere stata fatta che a seguito dell'ingresso in tale stato, e così la condizione associata diventa vera.

Dopo aver memorizzato l'evento comunicato mediante l'invocazione di `handleTransition`, il resto del codice fa in modo che nell'oggetto notificato si scatenino le eventuali conseguenze, cioè l'istanziazione di nuove sottoattività o un cambiamento di stato. Queste due possibilità vengono controllate rispettivamente dall'esecuzione dei metodi `instantiateChildren` (illustrato nella sezione 13.4) e `changeState:`.

13.3.4 Esecuzione delle attività

Dopo aver illustrato il funzionamento del metodo che gestisce i cambiamenti di stato è facile capire come comincia l'esecuzione dell'attività. Si è detto che quando le assegnazioni delle persone ad un'attività sono state portate a termine, su di essa viene invocato il metodo `endOfAssignation` il quale a sua volta, come si vede nella figura 13.22, manda il messaggio `changeState:` passando come argomento lo stato `#waitingPreconditions`. Questa invocazione porta l'oggetto interessato nello stato suddetto, ma se le precondizioni sono soddisfatte, gli fa fare un'ulteriore transizione in `#executing`.

Come si vede nella figura 13.23, la transizione nello stato `#executing` provoca l'invocazione di `startExecution` dell'oggetto coinvolto, ed ha così inizio l'esecuzione.

Il metodo `startExecution` è definito per la classe `Task` come mostrato nella figura 13.25 e realizza le operazioni che devono essere fatte da qualsiasi attività quando inizia la propria esecuzione. Nelle sottoclassi di `Task` questo metodo deve essere ridefinito in modo da poter specificare in esso le azioni che l'attività in questione deve eseguire; in ogni caso il primo messaggio delle implementazioni nelle sottoclassi deve eseguire questa realizzazione del metodo.

```
startExecution
    "executes task"

    self createTemporalOutput.
    self reConnectInput.
    self restartChildren.
    self instantiateChildren
```

Figura 13.25. Metodo `startExecution` di `Task`

13.3.5 Gestione dinamica dei dati in ingresso ed in uscita

I dati che vengono utilizzati da un'attività e quelli che da essa vengono prodotti, le vengono associati mediante gli attributi `input` e `output` subito dopo la creazione dell'oggetto corrispondete. È la stessa procedura di istanziazione che si fa carico di ciò, invocando l'apposito metodo privato `connectIOOf` come descritto nella sezione 13.4.5.

I dati temporanei

Si è detto in precedenza che nel soggetto `Data` esistono delle classi utilizzate per modellizzare gli oggetti software creati temporaneamente per gestire il processo e non fanno parte del prodotto di quest'ultimo. Un tipico esempio è la classe `FeedbackDocument` le cui istanze sono utilizzate per documentare il fallimento di attività. Gli oggetti di `Task` che prevedono la produzione di istanze di `TemporarySwObject` le creano all'inizio dell'esecuzione mediante l'invocazione di `createTemporalOutput`.

L'eliminazione delle istanze dei dati temporanei è compito dell'attività stessa che le ha prodotte ed è fatta nel momento in cui non sono più di nessuna utilità. Ad esempio per quanto riguarda gli oggetti `FeedbackDocument`, l'eliminazione avviene quando viene decretata la terminazione con successo dell'attività (sezione 13.3.9).

I dati in ingresso

Gli oggetti `Data` in ingresso alle attività sono individuati dalla connessione di istanza `input`. Questa è inizializzata al momento della creazione dell'attività, però va aggiornata perché possono esserci stati cambiamenti dovuti alla creazione di dati temporanei che al momento dell'istanziazione non c'erano ancora, o alla variazione della struttura del prodotto. Per questo motivo, ogni volta che inizia l'esecuzione di un'attività, questa invoca il metodo privato `reConnectInput`. Esso non fa altro che ricostruire la connessione `input` in modo analogo a quello presentato nella sezione 13.4.5 in cui si descrive la connessione dei dati come avviene subito dopo la creazione della nuova attività.

13.3.6 Risecuzione delle sottoattività

Nel codice del metodo `startExecution` riportato nella figura 13.25, si ha l'invocazione del metodo `restartChildren` per ricominciare l'esecuzione delle sottoattività. Infatti `startExecution` è eseguito ogni volta che si ha una transizione nello stato `#executing` ed è quindi possibile che siano presenti sottoattività che devono essere nuovamente eseguite.

```
restartChildren
   | tmp flag |
   tmp := Semaphore new.
   flag := false.
   Children everyChildrenOf:  self class do:  [:childClass |
childClass predecessorsNumber isZero ifTrue:  [(children
instancesOAOfClass:  childClass)
            do:
               [:childId |
               flag := true.

               [childId again.
               tmp signal] fork]]].
   flag
      ifTrue:
         [self signal.
         tmp wait.
         self wait]
```

Figura 13.26. Metodo `restartChildren` di `Task`

Come già detto in precedenza, la riesecuzione delle sottoattività deve comunque rispettare i vincoli di precedenza tra esse, e ciò è possibile utilizzando per la loro riattivazione il codice riportato nella figura 13.26. Si considerano tutte le sottoattività per cui il modello non preveda predecessori, cioè per cui non si hanno classi connesse da `Predecessors`. Agli oggetti corrispondenti a queste sottoattività viene inviato il messaggio `again` in modo che, se esse hanno finito la loro esecuzione, questa sia ricominciata. La riesecuzione delle sottoattività senza predecessori provoca la riattivazione delle attività che ne sono successori, così che tutte le sottoattività vengono rieseguite secondo l'ordine stabilito.

Si può notare che l'esecuzione di `again` su ciascuna sottoattività è affidata ad un processo indipendente, però la continuazione dell'esecuzione del padre non può continuare fino a che almeno una delle sottoattività non abbia aggiornato il proprio stato. Infatti, se l'esecuzione dell'attività composita consiste solamente nella gestione delle sottoattività, essa passa direttamente nello stato `#handlingChildren`. In tal caso, se almeno una sottoattività non è stata rieseguita, si hanno tutte le condizioni per transitare in `#completed`. Per questo motivo, ogni processo che abbia finito di eseguire l'`again`, invoca il metodo `signal` sul semaforo `tmp`. Invece `restartChildren`, se ha inviato almeno un messaggio `again`, esegue una `wait` su `tmp`, in modo che il processo corrispondente resti bloccato fino a che l'esecuzione di `again` su almeno una delle sottoattività non sia terminata.

Il metodo `again`

Il metodo `again` viene invocato per rieseguire le sottoattività di un oggetto `Task` che comincia l'esecuzione, oppure per rieseguire gli oggetti `Task` connessi mediante `waitingFeedback` a causa del fallimento di un'attività.

```
again
   "reactivate the task"

   self critical:  [state = #executing
      ifTrue:  [toBeRestarted := true]
      ifFalse:  [self changeState:  #waitingPreconditions]]
```

Figura 13.27. Metodo `Again` di `Task`

Nella figura 13.27 è riportato il codice di `again` che, trattandosi di un metodo pubblico, è racchiuso in una regione critica. Se lo stato in cui si trova l'oggetto è `#executing`, in accordo con quanto detto nella sezione 10.5.3, l'evento viene memorizzato nell'attributo `toBeRestarted` perché quest'informazione sia utilizzata da `changeState:` al momento opportuno. Se l'oggetto è in un altro stato, viene fatto transitare in `#waitingFeedback` da cui, appena possibile, passerà in `#executing` per ricominciare l'esecuzione.

13.3.7 Interazione con l'utente

Durante la sua esecuzione, un'attività può avere necessità di interagire con il proprio responsabile. Questo è fatto mediante opportune interfacce con cui, in modo assolutamente non controllato dal processo, l'utente è in grado di fornire informazioni. Quando questi segnala di avere terminato, l'attività interessata ne è informata così che possa continuare la propria esecuzione servendosi delle nuove informazioni.

Comunicazione con l'agenda

Il caso più semplice e più frequente di interazione, è la richiesta della disponibilità dell'utente a svolgere determinate azioni. Ciò è fatto aggiungendo una nuova voce nell'agenda della persona interessata mediante l'invocazione del metodo `askAttentionFor:description:`. Nella figura 13.28 è riportata l'implementazione di tale metodo che invia un messaggio al responsabile dell'attività, in modo che questo inserisca il nuovo elemento nella lista visualizzata dall'interfaccia a lui associata. Quindi il processo che ha eseguito queste istruzioni viene messo in attesa sul semaforo identificato dall'attributo `userAction` di `Task`. Si può notare che prima di inviare il messaggio `wait` a tale semaforo, si ha l'invocazione di `signal` su `self` in modo da permettere che sull'oggetto `Task` possano essere invocati altri metodi.

```
askAttentionFor: aString1 description: aString2
   "ask responsible attention"

   responsible
      recordActivity:  aString1
      from:  self
      description:  aString2.
   self signal.
   userAction wait.
   self wait
```

Figura 13.28. Metodo `askAttentionFor:description:` di `Task`

Quando l'utente comunica la propria disponibilità, l'istanza di `Agenda` interessata avvisa l'attività mandando il messaggio `gotUserAttention` che, come si vede nella figura 13.29 non fa altro che invocare il metodo `signal`

del semaforo `userAction`. Questo causa il risveglio del processo che, mentre eseguiva il corpo dell'attività, aveva richiesto l'interazione del responsabile. Tale processo innanzitutto invoca `wait` su `self` per assicurarsi di possedere l'unico flusso di controllo che sta eseguendo i metodi dell'oggetto, come si vede nell'implementazione di `askAttentionFor:description:`; quindi prosegue l'esecuzione.

```
gotUserAttention
    "the Responsible is available selecting the activity from Agenda"

    userAction signal
```

Figura 13.29. Metodo `gotUserAttention` di `Task`

13.3.8 Esecuzione degli strumenti automatici

L'interazione con l'utente per portare a termine le attività non avviene solamente attraverso interfacce appositamente definite dal sistema, ma anche mediante strumenti automatici che permettono di creare e modificare il prodotto delle attività. Quali strumenti siano da eseguire è determinato dagli oggetti `Data` connessi mediante `input` ed `output`.

Poiché S^3 è un ambiente per la simulazione, l'esecuzione degli strumenti non avviene realmente, ma è solamente simulata. Questo non cambia nulla dal punto di vista della classe `Task` perché la gestione dell'esecuzione sarebbe in ogni caso affidata all'implementazione della classe `Tool`. La responsabilità di `Task` sta nell'implementazione dei meccanismi per determinare l'istanza di `Tool` su cui invocare il metodo `runOn:` in modo da imporne l'esecuzione sul dato passato come argomento.

Gli oggetti di `Task` dispongono del metodo `runTools` che permette di lanciare l'esecuzione degli strumenti automatici. Nella figura 13.30 si ha il codice di tale metodo che prima di tutto invoca `chooseTools` fino a che questo non restituisce il valore vero.

Il compito di `chooseTools` è quello di connettere ad ogni oggetto `Data` che deve essere prodotto dall'attività, un'istanza di `Tool` mediante la connessione `created/creator` seguendo l'algoritmo presentato nella sezione 9.4.1. Se è necessario avere l'interazione dell'utente, il metodo esegue l'apposita interfaccia riportata nella figura 13.31. L'interfaccia è visualizzata senza chiedere

```
runToolOBs
   "run appropriate tools on input and output data"

   [self chooseTools]
      whileFalse:  [self askAttentionFor:  'Choose tools for ' ,
self getName description:  'Make tool choice and go on performing
activity'].
   (Array with:  input with:  output)
      do:  [:receiver | receiver do:
            [:data |
            | tool |
            (tool := data provideTool) notNil ifTrue:  [tool runOn:
data]]]
```

Figura 13.30. Metodo `runTools` di `Task`

prima la disponibilità dell'utente, perché l'esecuzione degli strumenti automatici avviene solamente quando questi abbia assicurato la sua disponibilità ad interagire. Può accadere che la persona coinvolta decida di sospendere la selezione mediante il bottone `Cancel`; in tal caso `selectTools` restituisce il valore falso e in `runTools` è fatta un'iterazione del blocco che richiede la disponibilità dell'utente bloccando il processo fino a che tale disponibilità non viene accordata.

Quando `selectTools` ritorna il valore `true`, `runTools` passa in rassegna ad una ad una le istanze di `Data` connesse mediante `input` e mediante `output`. Ad ognuna di esse invia il messaggio `provideTool` per ricavare lo strumento che la può manipolare e su questo invoca il metodo `runTools` che simula l'esecuzione dello strumento automatico corrispondente.

13.3.9 Terminazione delle attività

La terminazione di un'attività può essere decretata da un comando del responsabile attraverso un'interfaccia di comunicazione, oppure può essere stabilita automaticamente in base a ciò che deve essere fatto dall'attività. In ogni caso l'esecuzione può avere successo, oppure richiedere la riesecuzione delle attività connesse mediante `waitingFeedback`.

La terminazione con successo viene sancita con l'invocazione del metodo `terminateExecution` il cui codice è riportato nella figura 13.32. Questo non

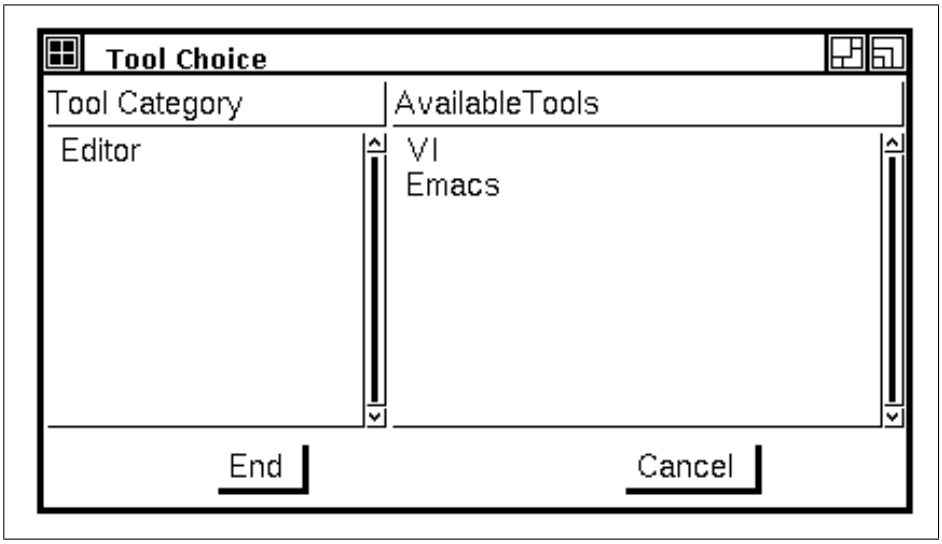

Figura 13.31. Interfaccia per la scelta degli strumenti automatici

fa altro che eliminare gli oggetti software prodotti temporaneamente dall'attività ed innnescare la transizione nello stato `#handlingChildren` mediante l'invocazione di `changeState:`.

terminateExecution

```
"declares the execution completed"

self removeTemporalOutput.
self changeState:  #handlingChildren
```

Figura 13.32. Metodo `terminateExecution` di `Task`

Il fallimento di un'attività è invece sancito con l'invocazione di `giveFeedback` (figura 13.33), che invia il messaggio `again` a tutti gli oggetti connessi mediante `waitingFeedback` e quindi innesca la transizione nello stato `#waitingPreconditions`, perché l'attività deve essere rieseguita non appena sia possibile. Si può notare che il cambiamento di stato è realizzato solo dopo che almeno una delle attività riattivate abbia reagito al messaggio `again`, cambiando quindi il proprio stato. Infatti, se la riattivazione non si è propagata fino ad i predecessori dell'attività in questione, quest'ultima, riportata

in `#waitingPreconditions` potrebbe essere erroneamente rieseguita immediatatente. Per ottenere l'attesa si utilizza un semaforo secondo lo stesso meccanismo usato da `restartChldren` e descritto nella sezione 13.3.6.

giveFeedback

```
"send feedback to waiters and children"

| tmp |
tmp := Semaphore new.
waitingFeedback do:  [:waiter |
   [waiter again.
   tmp signal] fork].
self signal.
tmp wait.
self wait.
self changeState:  #waitingPreconditions
```

Figura 13.33. Metodo `giveFeedback` di `Task`

13.4 Decomposizione delle attività

Il meccanismo per l'istanziazione delle sottoattività è uno dei più potenti tra quelli previsti da S^3. I metodi che realizzano questo meccanismo sono implementati nella classe `Task` e sono ereditati da tutte le sue sottoclassi senza dover essere ridefiniti grazie all'uso delle connessioni di livello classe. Tali metodi sono raccolti nel protocollo `children instantiation`; quando comincia l'esecuzione di un'istanza di `Task`, cioè questa passa nello stato `Executing`, viene invocato il metodo `instantiateChildren`, il quale procede all'istanziazione di tutte le sottoattività per cui ciò sia possibile in base ai principi dell'istanziazione incrementale.

Nella figura 13.34 è riportato il codice del metodo `instantiateChildren`. Mediante la connessione implementata dalla variabile di classe `Children`, che è visibie a tutte le istanze, si ricavano le classi delle sottoattività. Ognuna di queste, individuata dal parametro del blocco `childClass`, è usata come argomento nell'invocare un metodo privato in grado di stabilire se sia possibile creare istanze dell'attività considerata in base alle condizioni che vincolano l'istanziazione. Il metodo da utilizzare a questo scopo è diverso a seconda

```
instantiateChildren
   "instantiates children if possible"

   Children childrenAndNumberOf: self class do: [:childClass :num
| (num isZero
            ifTrue: [self isPossibleToInstantiateChildrenOfClass:
childClass]
            ifFalse: [self
isPossibleToInstantiateFixedChildrenOfClass: childClass])
            ifTrue: [num isZero
                  ifTrue: [self instantiateChildrenOfClass:
childClass]
                  ifFalse: [self instantiate: num childrenOfClass:
childClass]]]
```

Figura 13.34. Metodo `instantiateChildren` di `Task`

che il numero di possibili istanze sia noto a priori, o determinabile solamente al momento dell'istanziazione. Questa informazione è data dall'intero memorizzato nell'istanza di `ChildrenClassRelation` in corrispondenza della sottoattività in questione ed è stabilita dal progetto. Tale numero viene assegnato al parametro `num` del blocco passato come argomento al metodo `childrenAndNumberOf:do:`, e se è zero indica che il numero di istanze eventualmente necessarie non è noto staticamente a priori.

Indipendentemente dal metodo invocato per controllare se è possibile creare nuove istanze di una certa classe, il valore ritornato è un'istanza di `Boolean` a cui è inviato il messaggio `ifTrue:` con il blocco da eseguire se l'istanziazione è possibile. Questo blocco contiene l'invocazione al metodo che realizza effettivamente l'istanziazione, ma esso è diverso a seconda che `num` sia zero oppure no.

13.4.1 Controlli per l'istanziazione di sottoattività in numero noto

Nella figura 13.35 è riportato il metodo `isPossibleToInstantiateFixedChildrenOfClass` utilizzato per determinare se sia possibile istanziare sottoattività di una classe di cui sia noto a priori quanti oggetti siano necessari.

Quando è possibile l'istanziazione delle attività in numero noto a priori,

```
isPossibleToInstantiateFixedChildrenOfClass: childClass
   "test if is possible to instantiate a fixed number of children
of class childClass"

   | conditioningData |
   (children instancesOfClass:  childClass) isEmpty not ifTrue:
[↑false].
   conditioningData := Set new.
   childClass inputOutput do:  [:dataClass | (self
iOConditioningData instancesOfClass:  dataClass)
         do:  [:data | conditioningData addAll:  (data
conditioningDataFrom:  self)]].
   ↑(self isPossibleToInstantiateChildOfClass:  childClass
withConditioningData:  conditioningData)
```

Figura 13.35. Metodo `isPossibleToInstantiateFixedChildren Of-Class` di `Task`

tutti gli oggetti necessari sono creati e legati al padre mediante la connessione `children`. Dunque una prima condizione da verificare è che non ci sia nessuna istanza della classe passata come argomento dell'invocazione tra gli oggeti conessi mediante `children`. Invocando il metodo `instancesOfClass:` su `children` si ottiene un oggetto `Set` contenente tutte le istanze di `childClass` che sono sottoattività dell'oggetto che sta eseguendo il metodo; se tale insieme non è vuoto non serve creare nuove istanze della classe in esame ed allora è restituito il valore `false`.

Una seconda condizione che vincola l'istanziazione è che gli eventuali predecessori dell'attività modellizzata da `childClass` siano tutti nello stato `#completed`; questo controllo è realizzato dal metodo privato `isPossibleTo-InstantiateChildOfClass:withConditioningData:` il cui valore di ritorno è restituito al chiamante.

13.4.2 Determinazione dei predecesori di un'attività

La connessione `Successors/Predecessors` di livello classe fornisce informazioni solo riguardo al tipo di attività (classe) che condiziona l'esecuzione, ma per comprendere se sia possibile effettuare un'istanziazione, è necessario identificare gli oggetti che effettivamente la condizionano. Dunque le

istanze che devono essere in stato `#completed` prima di poter istanziare le attività di una certa classe vanno cercati tra le istanze delle classi connesse da `Predecessors`.

Utilizzo della gerarchia

Un'altra informazione importante nella ricerca dei predecessori si ha da una delle ipotesi fatte durante la progettazione e presentata nella sezione 10.2:

> le attività che influenzano l'istanziazione e l'esecuzione di un'altra attività (cioè i suoi predecessori), e questa stessa attività, devono provenire dalla decomposizione di un'unica attività composta.

Questa ipotesi deriva dal fatto che anche nella coordinazione delle attività ci si basa su principi di tipo gerarchico.

Utilizzo dei dati

Per l'implementazione dell'esempio di processo software, ma più in generale di qualsiasi altro processo, è necessario fare un'ipotesi aggiuntiva perché non è detto che tutte le attività ottenute secondo il criterio definito fino ad ora, debbano effettivamente condizionare l'attività in via di istanziazione. Ad esempio la compilazione di un sorgente non può essere eseguita fino a che tutti i 'file' che vengono utilizzati non sono stati scritti, quindi fino a che le attività che li producono non sono in stato `#completed`. Questo non significa che *tutte* le istanze di `Edit` devono essere nello stato `#completed`, ma solo quelle che producono oggetti `Data` che interessano l'istanza di `Compile` in questione. Allora è possibile introdurre un nuovo criterio di selezione delle attività che influenzano l'istanziazione di un'altra, basato sui dati che le istanze di `Task` manipolano.

Le relazioni tra gli oggetti di `Data` introdotte nella sezione 13.2.2 permettono di stabilire quali attività si condizionino le une con le altre. In particolare i predecessori un'attività `T1` si devono cercare tra le attività `Ti` che producono dati che condizionano i dati utilizzati da `T1`.

Applicazione dei due criteri

I due criteri per la determinazione dei predecessori sono applicati nei metodi che implentano l'istanziazione delle sottoattività. Esiste un metodo che

controlla lo stato dei predecessori per determinare se sia possibile creare una nuova attività (`isPossibleToInstantiateChildOfClass:withConditioning Data:`). A questo metodo si deve passare come parametro un insieme contenente i dati che condizionano i dati che saranno manipolati dalla nuova attività. La determinazione di questo insieme viene fatta dal metodo `isPossibleToInstantiateFixedChildrenOfClass:` riportata nella figura 13.35. Innanzi tutto si determinano le sottoclassi di `Data` cui appartengono gli oggetti in ingresso ed in uscita all'attività da creare; ciò è fatto invocando il metodo `inputOutput` sulla classe dell'attività. Le istanze di queste classi che dovranno effettivamente essere manipolate dall'attività da istanziare, saranno in qualche modo correlate ai dati in ingresso o in uscita all'attività che sta eseguendo l'istanziazione, cioè si troveranno all'interno dell'insieme restituito dall'invocazione del metodo `iOConditioningData` su `self`.

iOConditioningData

```
"return data conditioning I/O"

| tmp |
tmp := Set new.
(Array with: input with: output)
   do: [:sender | sender do:
      [:data |
         tmp addAll: (data conditioningDataFrom: data).
         tmp add: data]].
↑tmp
```

Figura 13.36. Metodo `iOConditioningData` di `Task`

Il metodo `iOConditioningData`, la cui implementazione è riportata nella figura 13.36, costruisce l'insieme da ritornare invocando il metodo `conditioningData From:` su tutte le istanze di `Data` connesse mediante `input` o `output` all'attività che sta realizzando l'istanziazione.

Dall'istanza di `Set` che `iOConditioningData` restituisce, il metodo `isPossibleToInstantiateFixedChildrenOfClass:` estrae gli oggetti delle classi che sono manipolate dall'attività che si vuole istanziare mediante il metodo `instancesOfClass:`. Di ognuno di questi oggetti si cercano i dati legati dalla relazione di condizionamento, invocando su di essi il metodo

`conditioningDataFrom:` e le istanze così ottenute vengono alfine inserite nella variabile temporanea `conditioningData`, che sarà usata come parametro per l'invocazione del metodo `isPossibleToInstantiateChildOf-Class:withCon ditioningData`. Il codice di quest'ultimo, riportato nella figura 13.37, contiene i messaggi utilizzati per comprendere se i predecessori dell'attività da istanziare sono nello stato `#completed`.

```
isPossibleToInstantiateChildOfClass: childClass
   "check if all predecessors are completed"

   childClass
      predecessorsDo:
         [:predClass |
         | tmp |
         tmp := Set new.
         data do:
            [:each |
            | temp |
            (temp := each predecessorTasksOfClass:  predClass) isNil
ifTrue:  [↑false].
            tmp addAll:  (self onlyChildrenAmong:  temp)].
         (tmp isEmpty
            ifTrue:  [(children allCompletedOfClass:  predClass)
                  & (children instancesOfClass:  predClass) isEmpty
not]
            ifFalse:  [children allCompleted:  tmp])
            ifFalse:  [↑false]].
   ↑true
```

Figura 13.37. Metodo `isPossibleToInstantiateChildOf Class:with-ConditioningData` di `Task`

In questo metodo si ha l'effettiva applicazione dei criteri presentati precedentemente per la determinazione dei predecessori di un'attività che si vuole istanziare. Quindi l'intento è di ottenere tutte le istanze delle classi collegate dalla connessione `Predecessors` alla classe dell'attività da istanziare che

1. siano sottoattività della classe che sta eseguendo l'istanziazione (`self`);
2. siano produttrici di almeno uno degli oggetti contenuti in `data`, cioè il parametro contenente i dati che si sono stimati essere in qualche modo

correlati a quelli che dovranno essere prodotti dalla nuova attività.

Le classi dei predecessori di `childClass` sono passate in rassegna inviando a quest'ultima il messaggio `predecessorsDo:` ed assegnandole di volta in volta al parametro `predClass`. Quindi, su ognuno degli oggetti contenuti in `data`, è invocato il metodo `predecessorTasksOfClass:`; questo ritorna le istanze della classe passata come argomento che sono connesse mediante `productors` al dato in questione. Il nome utilizzato per il selettore di questo metodo è stato scelto in modo da aumentare la leggibilità del codice, anche se non descrive ciò che il metodo restituisce in generale, ma solo in questa sua particolare applicazione.

Dell'insieme di attività così ottenuto, si inseriscono nella variabile temporanea `tmp` solamente gli elementi che sono sottoattività di `self`; questi sono individuati invocando il metodo `onlyChildrenAmong:`.

A questo punto `tmp` contiene tutti i predecessori che si sono ricavati per la classe `predClass` in esame e si possono valutare due situazioni che portano ad escludere la possibilità di creare istanze di `childClass`:

1. `tmp` contiene degli oggetti che non sono tutti nello stato `#completed` (controllato con il messaggio `allCompleted:` inviato a `children`);

2. `tmp` è vuoto, ma ci sono sottoattività della classe `predClass` che non sono nello stato `#completed`.

Nei suddetti due casi viene restituito `false`; diversamente si ripete il procedimento per tutti i valori assunti da `predClass` ed alla fine, cioè dopo aver considerato tutte le classi di predecessori, viene ritornato `true`.

Dall'implementazione del meccanismo per l'identificazione dei predecessori di un'attività facendo uso dei dati, si vede come gli oggetti effettivamente inidividuati come predecessori dipendano dalle relazioni tra i dati. In particolare alcune relazioni tra attività dipendono dalle implementazioni del metodo `conditioningDataFrom:` nelle singole sottoclassi di `Data`; nella sezione 13.2.1 è riportata ad esempio la ridefinizione per `SourceFile`. Ciò significa che nel definire gli elementi che possono far parte della struttura del prodotto, cioè le sottoclassi di `Data`, è fondamentale chiarire bene anche le relazioni di condizionamento e di correlazione che si hanno tra le istanze. Questo non può essere fatto concentrando l'attenzione solamente sul soggetto `Data`, ma si devono tenere presenti anche le attività che saranno chiamate a manipolare questi dati.

13.4.3 Istanziazione di sottoattività in numero noto

Il metodo `instantiateChildren`, come si vede dalla figura 13.34, dopo aver ottenuto risposta positiva dall'invocazione di `isPossibleToInstantiateFixedChildrenOfClass:`, passa all'istanziazione degli oggetti della classe per cui si è fatto il controllo mandando a `self` il messaggio `instantiate:childrenOfClass:` che ha come argomento il numero di istanze da creare e la classe interessata.

```
instantiate: number childrenOfClass: childrenClass
   "instantiates number children of the class childrenClass"

   number
      timesRepeat:
         [| newChild |
         children add:  (newChild := childrenClass new:  self).
         self connectIOOf:  newChild.
         newChild associatedTools:  associatedTools.
         self connectSiblingsOf:  newChild.
         self assignFor:  newChild]
```

Figura 13.38. Metodo `instantiate:childrenOfClass:` di `Task`

Nella figura 13.38 si ha il codice di `instantiate:childrenOfClass:` che consta nella ripetizione di un blocco per un numero di volte pari al numero di sottoattività da istanziare. Ad ogni ripetizione è creata una nuova istanza della classe passata nell'argomento `childrenClass`; questa nuova istanza viene connessa come sottoattività mediante `children`, e quindi le sono connessi i dati da utilizzare e da produrre, gli stessi strumenti automatici del padre, le attività che la devono precedere o a cui deve notificare il proprio fallimento ed infine le persone che avrà a disposizione. I metodi che permettono di realizzare queste azioni sono descritti nelle sezioni successive.

13.4.4 Controlli per l'istanziazione di sottoattività in numero non noto

Il metodo `instantiateChildren` (figura 13.34), per controllare se sia possibile istanziare sottoattivtà di classi per cui non è noto a priori il numero di

oggetti necessario, invoca il metodo `isPossibleToInstantiateChildrenOfClass:` la cui implementazione è riportata nelle figura 13.39.

```
isPossibleToInstantiateChildrenOfClass: childClass
   "test if is possible to instantiate children of class childClass"

   | inputConditioningData |
   inputConditioningData := Set new.
   childClass input do:  [:dataClass | (self iOConditioningData
instancesOfSubclassOf:  dataClass)
         do:  [:data | inputConditioningData addAll:  (data
conditioningDataFrom:  self)]].
   childClass output do:  [:dataClass | (self iOConditioningData
instancesOfSubclassOf:  dataClass)
         do:  [:dataObj | dataObj productors isEmpty
               ifTrue:
                  [| conditioningData |
                  conditioningData := inputConditioningData copy.
                  conditioningData addAll:  (dataObj
conditioningDataFrom:  self).
                  (self isPossibleToInstantiateChildOfClass:
childClass withConditioningData:  conditioningData)
                     ifTrue:  [↑true]]]].
   ↑false
```

Figura 13.39. Metodo `isPossibleToInstantiateChildrenOfClass:` di `Task`

Una prima differenza tra questo metodo e quello analogo utilizzato con le classi per cui si conosce il numero di istanze desiderate, sta nel fatto che non si può imporre che l'istanziazione avvenga solo qualora non esistano già altre istanze della classe in esame. Un semplice esempio che documenta il fatto che questa condizione non è accettabile, è il caso delle attività della classe `Compile` che devono essere istanziate a mano a mano che le corrispondenti attività di scrittura di codice terminano, cioè con grande probabilità in istanti differenti.

Esiste comunque un controllo che può essere considerato l'analogo di quello appena descritto per le sottoattività che richiedono un numero noto di istanze. Infatti l'istanziazione di una sottoattività del tipo preso in considerazione (`childClass`) ha senso solo se esiste un'istanza di una sottoclasse di

`Data` legata dalla connessione di livello classe `Ouput`, che non abbia ancora un produttore. Questa istanza di `Data` viene ovviamente cercata solamente tra i dati in qualche modo correlati all'attività che sta esegiendo la decomposizione e che sono ricavati mediante il metodo `iOConditioningData` descritto nella sezione 13.4.2.

Per ogni oggetto `dataObj` che non ha produttori (perché l'invocazione di `productors` restituisce un insieme vuoto), si ricava l'insieme di dati che lo condizionano e lo si unisce all'insieme `inputConditioningData` ottenendo `conditioningData`. `inputConditioningData` si è ottenuto precedentemente a partire dalle istanze ritornate dall'invocazione `iOConditioningData` su `self`, le quali appartengono a sottoclassi di `Data` collegate a `childClass` dalla connessione di livello classe `Input`. L'insieme unione, `conditioningData`, è usato come argomento dell'invocazione di `isPossibleToInstantiate-ChildOfClass:withConditioningData:` il cui valore di ritorno permette di stabilire se è possibile istaziare almeno un'attività di `childClass`. Se ciò è vero viene restituito `true`, diversamente si ripete il ciclo per altri valori di `dataObj` fino a che, esauritili tutti, è ritornato il valore `false`.

13.4.5 Connesione dei dati ad una nuova sottoattività

Nella figura 13.40 è riportato il codice del metodo `connectIOOf:` che è utilizzato da un'attività che si decompone per connettere alle sue sottoattività i dati che le riguardano. Il metodo è diviso in due parti simmetriche, una delle quali costruisce la connessione `input` e l'altra la connessione `output` della nuova attività. In ognuna delle due parti vengono ricavate le sottoclassi di `Data` individuate dalle corrispondenti relazioni `Input` ed `Output` della classe di `newChild`, e le si assegna ad una ad una alla variabile temporanea `dataClass`. L'invocazione di `iORelatedData` consente di ricavare tutti i dati correlati a quelli in ingresso o in uscita all'attività che si sta decomponendo. Gli oggetti di questo insieme che sono istanze di `dataClass` vengono inseriti nella connessione di istanza che si sta creando.

13.4.6 Metodo `connectSiblings`

Dopo che una sottoattività è stata creata, è necessario connetterle gli opportuni oggetti di `Task` mediante `predecessors` e `waitingFeedback`. Si è già

```
connectIOOf: newChild
   "connect IO for newChild.
   The method connects instances of classes specified by Input and
   Output newChild's class relations that are connected by input
   or output Instance Connection of self"

   | data |
   data := InstanceConnection new.
   newChild class input do:  [:dataClass | data addAll:  (self
iORelatedData instancesOfClass:  dataClass)].
   newChild inputIs:  data.
   data := InstanceConnection new.
   newChild class output do:  [:dataClass | data addAll:  (self
iORelatedData instancesOfClass:  dataClass)].
   newChild outputIs:  data
```

Figura 13.40. Metodo `connectIOOf:` di `Task`

detto in precedenza che tali oggetti devono essere scelti tra le attività generate dalla stessa attività che si sta decomponendo perché si vuole realizzare una gestione gerarchica della coordinazione tra le attività.

Il compito di stabilire le connessioni `predecessors` e `waitingFeedback` di una nuova sottoattività è assegnato al metodo `connectSiblingsOf:` che è riportato nella figura 13.41. Gli oggetti `Task` candidati ad essere connessi sono ricavati mediante l'invocazione di `conditioningTasks` (sezione 13.2.2) su ciascuno dei dati connessi per mezzo di `output`. Gli elementi dell'insieme ottenuto che sono sottoattività dell'oggetto che sta eseguendo la decomposizione, sono inseriti in `conditioningTasks`.

Il resto del metodo considera tutte le sottoclassi di `Task` individuate dalle connessioni `Predecessors` e `WaitingFeedback` di livello classe. Per ognuna di esse ricava le istanze contenute in `conditioningTasks` e le inserisce nella opportuna connessione di livello istanza. Se per una classe di predecessori o di attività che devono ricevere notifica di fallimento non ci sono istanze in `conditioningTasks`, vengono connesse tutte le sottoattività appartenenti a tale classe.

```
connectSiblingsOf: newChild
   "connect predecessors and feedback waiters"

   | conditioningTasks |
   conditioningTasks := Set new.
   newChild output do:  [:data | conditioningTasks addAll:  data
conditioningTasks].
   conditioningTasks := self onlyChildrenAmong:  conditioningTasks.
   newChild class predecessorsDo:
      [:predClass |
      | tmp |
      (tmp := conditioningTasks instancesOfClass:  predClass) size
isZero
         ifTrue:  [(children instancesOfClass:  predClass)
               do:  [:predId | newChild addPredecessor:  predId]]
         ifFalse:  [tmp do:  [:task | newChild addPredecessor:
task]]].
   newChild class feedbackWaitersDo:
      [:waiterClass |
      | tmp |
      (tmp := conditioningTasks instancesOfClass:  waiterClass) size
isZero ifFalse:  [tmp do:  [:aTask | newChild addFeedbackWaiter:
aTask]]
         ifTrue:  [(children instancesOfClass:  waiterClass)
               do:  [:waiterId | newChild addFeedbackWaiter:
waiterId]]]
```

Figura 13.41. Metodo `connectSiblingsOf:` di `Task`

13.4.7 Assegnazione delle persone alle sottoattività

Quando un'attività si decompone in sottoattività, deve assegnare delle persone alle proprie sottoattività; le fasi iniziali di questa operazione sono compito del metodo `assignFor:` che riceve nell'argomento `childId` l'attività per cui le assegnazioni devono essere fatte.

Come si vede dalla figura 13.42 in primo luogo viene creato un insieme in cui sono inseriti i ruoli che devono essere ricoperti dai responsabili delle sottoattività di `childId`. Quindi, per ognuno di questi ruoli (si tratta di classi), si ricavano le persone assegnate all'attività che si sta decomponendo. Se c'è una sola persona, questa è direttamente connessa a `childId`, diversamente

è necessario che il responsabile dell'attività che si sta decomponendo scelga quali persone assegnare tra quelle candidate; questo fatto è memorizzato in `needAssignTasks`. Un procedimento analogo è seguito per l'assegnazione del responsabile di `childId`.

Se è necessaria l'interazione con l'utente, la si affida ad un'istanza di `AssignTasks` che è sottoattivtà dell'oggetto `Task` che sta eseguendo il metodo. Può essere che una sottoattività `AssignTasks` sia già presente tra quelle connesse mediante `children`; se così è, non ne viene creata un'altra, ma è riattivata l'istanza esistente, invocando su di essa il metodo `again`, dopo averla connessa a `childId` mediante `toAssign`. Diversamente viene istanziata una sottoattività `AssignTasks` mediante l'opportuno metodo descritto nella sezione 13.5.

Se non è necessaria l'interazione con una persona e le assegnazioni possono essere fatte tutte automaticamente, viene invocato il metodo `endOfAssignation` su `childId` facendolo passare in `#waitingPreconditions`.

13.5 La classe `AssignTasks`

La definizione di `AssignTasks` riportata nella figura 13.43 mostra la presenza dell'attributo `toAssign` che è utilizzato per connettere tutte le attività a cui è necessario assegnare delle persone.

13.5.1 Creazione delle istanze

La creazione degli oggetti `AssignTasks` avviene mediante l'invocazione del metodo di classe `new:responsible:assign:` la cui implementazione è riportata nella figura 13.44. Questo metodo riceve come argomenti l'attività che ha generato la nuova istanza (come il metodo di creazione delle istanze di qualsiasi altra sottoclasse di `Task`), il responsabile e l'oggetto `Task` per cui occorre fare l'assegnazione. I tre argomenti sono passati al metodo di inizializzazione che si fa carico di assegnarli agli opportuni attributi.

Poiché il responsabile è lo stesso dell'attività che ha generato la nuova istanza di `AssignTasks` e dal momento che questo tipo di attività non prevede predecessori, l'esecuzione può cominciare immediatamente dopo aver inizializzato gli attributi.

13.5.2 L'esecuzione

Come per ogni altra attività, l'esecuzione comincia con l'invocazione del metodo `startExecution` riportato nella figura 13.45. Se non ci sono oggetti connessi mediante `toAssign`, viene invocato `terminateExecution` che porta l'attività nello stato `#completed` (non ha sottoattività di cui attendere il completamento). Diversamente viene richiesta la disponibilità dell'utente ad interagire e, quando questi la concede, è eseguita l'interfaccia che permette di selezionare le persone da assegnare.

Nella figura 13.46 è riportata la finestra visualizzata da `AssignTasks` per consentire al responsabile di selezionare le persone da assegnare alle varie attività. L'interfaccia è composta da cinque liste in cui l'utente può selezionare degli elementi.

La lista a sinistra contiene l'elenco delle attività per cui è necessario operare delle assegnazioni che richiedono scelte da parte di una persona. Selezionando una delle voci nella lista di sinistra, nella lista centrale appaiono i ruoli che devono essere ricoperti dalle persone assegnate all'attività in questione perché sono richiesti per i responsabili delle sue sottoattività. Selezionando uno degli elementi della lista dei ruoli, nella lista in alto a destra compaiono le persone disponibili a ricoprire tale ruolo. Gli oggetti corrispondenti sono ricavati interrogando l'istanza della classe che modellizza il ruolo in questione che è connessa al padre di `AssignTasks`. Infatti le istanze di `AssignTasks` realizzano assegnazioni per attività che hanno il loro stesso padre e quindi le persone candidate vanno cercate proprio tra quelle a disposizione del padre comune.

Selezionando un elemento nella lista delle persone disponibili, esso viene trasferito nella lista sottostante che contiene l'elenco degli utenti che si voglio assegnare nel ruolo specificato. Per annullare una scelta fatta è sufficiente selezionare la corrispondente voce nella lista centrale, e la persona corrispondente viene nuovamente trasferita nella lista superiore.

La lista in basso a destra contiene, quando occore, l'elenco delle persone candidate ad essere assegnate come responsabile dell'attività selezionata. Per scegliere una persona è sufficiente selezionare l'elemento corrispondente e questo resterà evidenziato.

assignFor: childId

```
   |neededRoleClasses rolesToConnect playingPersons needAssignTasks|
   needAssignTasks := false.
   rolesToConnect := InstanceConnection new.
   neededRoleClasses := childId class childrenNeededRoles.
   neededRoleClasses do:
      [:roleClass |
      playingPersons := (assignedRoles instanceOfClass:  roleClass)
players.
      playingPersons size == 1
         ifTrue:  [rolesToConnect add:  (roleClass new:
playingPersons)]
         ifFalse:
            [needAssignTasks := true.
            rolesToConnect add:  (roleClass new:  InstanceConnection
new)]].
   childId class responsibleRole isNil
      ifTrue:  [childId responsibleIs:  responsible]
      ifFalse:
         [playingPersons := (assignedRoles instanceOfClass:  childId
class responsibleRole) players.
         playingPersons size == 1
            ifTrue:  [playingPersons do:  [:pers | childId
responsibleIs:  pers]]
            ifFalse:  [needAssignTasks := true]].
   childId connectRoles:  rolesToConnect.
   needAssignTasks
      ifTrue:
         [| assigner |
         (assigner := children instanceOfClass:  AssignTasks) isNil
            ifFalse:
               [assigner assign:  childId.
               [assigner again] fork]
            ifTrue:  [children add:  ((AssignTasks
                     new:  self
                     responsible:  responsible
                     assign:  childId)
                     connectRoles:  assignedRoles)]]
      ifFalse:  [[childId endOfAssignation] fork]
```

Figura 13.42. Metodo `assignFor:` di `Task`

```
Task subclass:  #AssignTasks
   instanceVariableNames:  'toAssign '
   classVariableNames:  ''
   poolDictionaries:  ''
   category:  'Task'
```

Figura 13.43. Definzione della classe `AssignTasks`

```
new: father responsible: resp assign: taskId
   "creates a new instance of the task"

   ↑super new
      initialize:  father
      responsible:  resp
      assign:  taskId
```

Figura 13.44. Metodo `new:responsible:assign:` di `AssignTasks`

```
startExecution
   "start execution of task body"

   toAssign isEmpty
      ifTrue:  [self terminateExecution]
      ifFalse:
         [super startExecution.
         self askAttentionFor:  'Assign Tasks' description:
'Select responsibles for shown tasks'.
         self openSelectionInterface]
```

Figura 13.45. Metodo `startExecution` di `AssignTasks`

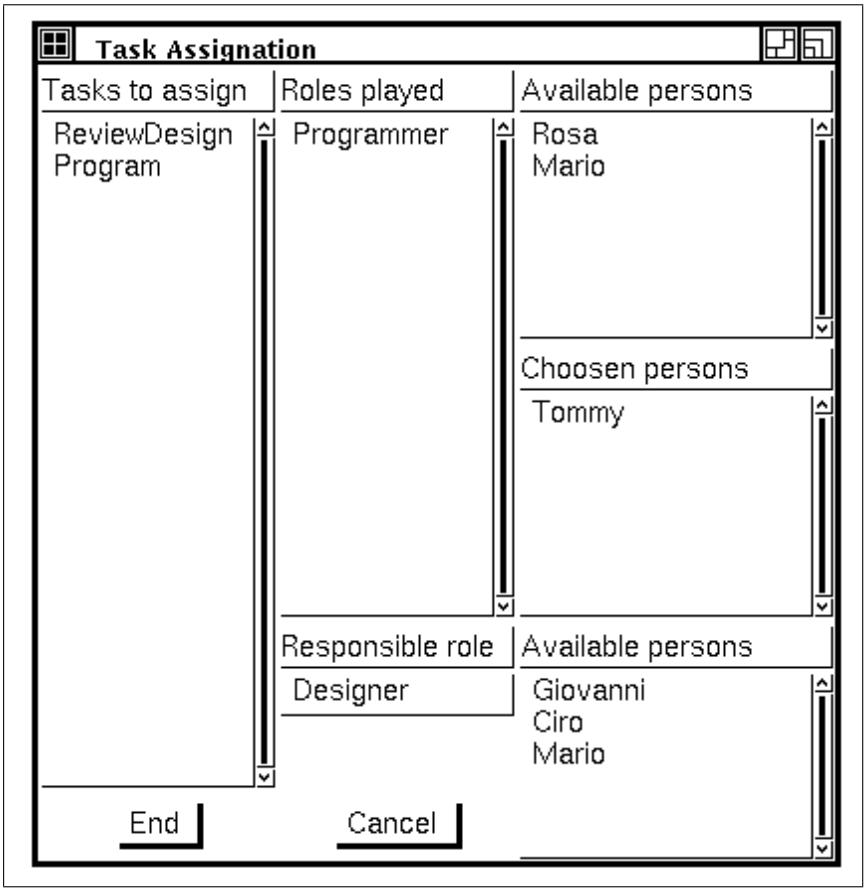

Figura 13.46. Interfaccia per l'assegnazione delle persone alle attività

14

Conclusioni

Il lavoro descritto in questa trattazione ha permesso di studiare alcune problematiche interessanti in prospettiva alla realizzazione di E^3. Inoltre ha aperto nuove tematiche di ricerca per approfondire ed ampliare il lavoro svolto.

14.1 Risultati conseguiti

Con questo lavoro, da un lato sono stati introdotti nuovi concetti e strumenti per la modellizzazione ad oggetti di processi software, dall'altro si sono date valutazioni su strumenti già esistenti e magari utilizzati in altri campi di applicazione.

Meta-processo proposto

Innanzitutto si è definito un modello di meta-processo, cioè la descrizione di un processo utilizzabile per la produzione dei processi software. Questo meta-processo si basa su quattro fasi principali:

1. `analisi e progettazione`
2. `implementazione`
3. `istanziazione ed esecuzione`
4. `manutenzione`.

I modelli di processo da simulare con S^3 possono essere realizzati seguendo questo modello di meta-processo che potrà eventualmente essere utilizzato anche per la modellizzazione dei processi in E^3.

Orientamento agli oggetti per PM

Si è potuto verificare che l'orientamento agli oggetti costituisce un efficace strumento per la modellizzazione dei processi software, soprattutto quando il formalismo utilizzato preveda una rappresentazione esplicita del livello classe e delle relazioni tra le entità che lo costituiscono. Questo aspetto del formalismo è indispensabile nella fase dell'`implementazione` di modelli di processo, ma se ne può fare a meno nella fase di `analisi e progettazione`. Se il linguaggio di implementazione dei modelli di processo consente una rappresentazione esplicita del livello classe, è possibile realizzare nelle classi più generali, meccanismi validi in tutto il modello che non devono essere ridefiniti nelle sottoclassi.

- Il formalismo proprio della metodologia Coad/Yourdon non presenta le caratteristiche appena dette, ed infatti è stato leggermente ampliato per una più efficace modellizzazione dell'esempio di processo.
- Le funzionalità messe a disposizione da Smalltalk-80 sono pienamente soddisfacenti dal punto di vista dei requisiti richiesti ad un formaliso ad oggetti per la modellizzazione dei processi software. Infatti Smalltalk è molto flessibile ed efficiente grazie alla sua uniformità, che consta nel fatto che tutto è gestito utilizzando oggetti e scambi di messggi tra essi.

Tecnica di modellizzazione ad oggetti proposta

La tecnica per la modellizzazione ad oggetti di processi software proposta in questa trattazione si basa sulle funzionalità messe a disposizione da S^3. Il vantaggio principale di questa tecnica sta nel fatto che i modelli di processo possono essere realizzati con qualsiasi metodologia di progettazione ad oggetti e richiedono soltanto di avere a disposizione uno schema predefinito di classi analogo a quello fornito da S^3.

Per capire la potenza della tecnica di modellizzazione bisogna tenere presenti alcuni punti fondamentali.

- In questa trattazione è stato illustrato un progetto la cui implementazione ha dato origine a S^3. L'ambiente PM descritto nel progetto in questione è costituito da un insieme di classi, collegate da connessioni, che sono in grado di comunicare tra loro e creare istanze. Anche le

istanze possono essere legate da connessioni e sono in grado di comunicare tra loro e con le classi.

- Il modello di un processo è un insieme di classi, alcune delle quali predefinite, che sono collegate da connessioni, di livello classe e di livello istanza;
- Il modello di processo eseguibile è l'implementazione di queste classi in un linguaggio eseguibile;
- Il processo in esecuzione è un insieme di oggetti, tra cui anche quelli rappresentanti le classi, che sono connessi tra loro e scambiano messaggi.

L'implementazione del modello di processo ottenuto dalla fase di progettazione, può essere realizzata mediante qualsiasi linguaggio di programmazione, anche non ad oggetti. L'importante è ottenere alla fine un insieme di entità, con le stesse caratteristiche tipiche degli oggetti, le quali comunicano tra di loro. L'implementazione del progetto dell'ambiente PM e dei modelli di processo risulta senza dubbio più semplice ed immediata utilizzando un linguaggio di programmazione ad oggetti, o qualsiasi altro formalismo che metta a disposizione entità analoghe a classi, oggetti, attributi e servizi, con gli stessi significati e funzionalità che sono associati ad essi dall'orientamento agli oggetti.

Da queste premesse risulta immediata l'espansione di S^3 ad un ambiente distribuito, perché è sufficiente allocare su macchine differenti i vari oggetti che compongono il modello di processo in esecuzione. Dando a questi oggetti la capacità di comunicare da una macchina all'altra, i modelli di processo risultano perfettamente eseguibili in un ambiente distribuito senza cambiare minimamente i principi su cui si basano il funzionamento di S^3 e la modellizzazione dei processi secondo la tecnica proposta.

Un problema di S^3 è la mancanza di uno strumento di supporto alla modellizzazione. Infatti la costruzione dei modelli secondo la tecnica di modellizzazione proposta, richiede la conoscenza dei meccanismi che stanno alla base dell'ambiente che la supporta. L'inserimento di una classe in un modello implica che essa venga opportunamente collegata ad un certo numero di altre classi che si possono individuare solo avendo una profonda conoscenza dell'ambiente e dei sui meccanismi. Questa limitazione può essere superata

realizzando strumenti automatici che assistano nella creazione dei modelli, e soprattutto nel riutilizzo delle classi predefinite.

DECdesign

Un ambiente per la modellizzazione di processi software deve fornire strumenti che aiutino nella modellizzazione e nell'attuazione del processo. DECdesign è uno dei candidati ad essere utilizzato nella fase del meta-processo che riguarda la progettazione del processo software. Inoltre DECdesign è candidato ad essere utillizzato anche durante l'attuazione del processo software, per la progettazione del prodotto finale.

DECdesign fornisce supporto alla progettazione secondo cinque differenti metodologie; l'utilizzo nell'abito di questo lavoro è limitato alla sola metodologia Coad/Yourdon, e quindi le valutazioni fatte riguardano esclusivamente questo ambito.

L'utilizzo di DECdesign per la progettazione presenta sicuramente dei vantaggi:

- supporto per la manipolazione dei simboli grafici che fanno parte della notazione proposta dalla metodologia Coad/Yourdon;
- facilitazioni nella navigazione delle varie viste di cui è costituito il progetto;
- rispetto della semantica dei tipi di relazioni tra le entità definite dalla metodologia Coad/Yourdon;
- rudimentale generazione di codice C++ a partire dal progetto.

DECdesign presenta però anche alcune lacune:

- le viste assumono dimensioni proppo grandi divenendo spesso difficili da visionare;
- i formati per la stampa dei progetti sono fissi ed alcune viste risultano pertanto suddivise su un numero così alto di pagine da non essere comprensibili;
- problemi nel tracciamento delle connessioni tra i simboli tipici del formalismo Coad/Yourdon;

- problemi degli 'editor' grafici che mancano di funzioni di grande utilità.

La versione attuale del prodotto non è adatta a gestire progetti di grandi dimensioni e quindi analogamente non si può utilizzare per la modellizzazione di processi che richiedano un gran numero di classi.

14.2 Sviluppi futuri

I risultati ottenuti si possono considerare una buona base per lo sviluppo di un ambiente PM ad oggetti completo e potente. Però ancora molto lavoro, ed in varie direzioni, resta da fare per la realizzazione di un tale ambiente.

Distribuzione Un ambiente per la modellizzazione dei processi deve essere indubbiamente un sistema distribuito in cui molte persone, lavorando su varie macchine, interagiscono con l'ambiente per portare avanti i processi di produzione del software. Non è particolarmente oneroso implementare il progetto di S^3 in modo da ottenere un ambiente PM distribuito, a patto di disporre di un adeguato supporto (vedi sezione precedente). Nell'ambito del progetto E^3 si stanno studiando alcune ipotesi su come realizzare la distribuzione sfruttando come supporto lo standard OMG o i meccanismi di comunicazione messi a disposizione dal sistema operativo UNIX.

Integrazione degli strumenti automatici Per passare da un ambiente di simulazione ad uno per l'attuazione dei processi software, è necessario essere in grado di eseguire gli strumenti automatici in modo controllato e distribuito. Queste tematiche sono state affrontate nel progetto E^3 studiando il supporto fornito in questo senso dagli ACA Service.

Strumenti per la modellizzazione dei processi Un efficiente ambiente PM deve fornire strumenti che facilitino al massimo la creazione dei modelli di processi software, dando supporto all'utente nell'utilizzo delle classi predefinite dal sistema. Inoltre gli strumenti di supporto alla modellizzazione secondo la tecnica proposta, devono consentire l'utilizzo dello schema messo a disposizione dall'ambiente, senza richiedere una conoscenza dettagliata dei meccanismi su cui l'ambiente si basa.

Integrazione con basi di dati È importante disporre di meccanismi per la memorizzazione ed il recupero dei dati prodotti dalle attività del processo, magari assicurandone il versionamento. Questo può essere garantito dall'utilizzo di una base di dati come substrato dell'ambiente PM.

L'integrazione con una base di dati fornirebbe supporto anche per la memorizzazione ed il riutilizzo dei modelli di processo e dello schema di classi predefinite che l'ambiente mette a disposizione per la creazione di nuovi modelli.

Evoluzione In questo lavoro non ci si è curati della possibilità di modificare i modelli, nè dell'impatto che eventuali modifiche apportate avrebbero sull'esecuzione dei processi. Questo è quindi un valido spunto di ricerca per la realizzazione di un ambiente PM vero e proprio perché l'evoluzione è un aspetto fondamentale nella modellizzazione dei processi software.

Oltre ai processi in esecuzione, e quindi i loro modelli, si deve essere in grado di modificare il meta-processo ed il meta-modello, cioè tutta la struttura per la realizzazione di modelli di processo.

Appendice A

DECdesign

Uno degli scopi del lavoro descritto in questa trattazione è la valutazione dell'efficienza di DECdesign come strumento di supporto alla progettazione ad oggetti. In questa appendice si dà una panoramica sul funzionamento di DECdesign, descrivendo gli strumenti e le facilitazioni che mette a disposizione del progettista. Infine si sono evidenziati i vantaggi e gli svantaggi che derivano dall'uso di questo strumento automatico.

A.1 Caratteristiche generali

Per la realizzazione del progetto dell'esempo di processo software proposto nella sezione 11.3 ci si è utilizzato DECdesign V2.0 [D.E92b] per il sistema operativo UTRIX 4.2.

Questo strumento automatico fornisce supporto alla progettazione secondo cinque differenti metodologie. Come già anticipato la modellizzazione del processo software è stata fatta utilizzando la metodologia Coad/Yourdon esposta in [CY91a] e [CY91b] ed implementata da DECdesign.

I documenti prodotti da questo strumento sono detti *librerie*, costituite da un insieme di *viste* ('*View*') che sono differenti a seconda della metodologia che si sta utilizzando. DECdesign fornisce allora una serie di funzionalità per la gestione delle librerie che sono comuni alle implementazioni di tutte le metodologie di progetazione.

DECdesign fornisce una serie di funzioni che permettono da una parte di effettuare queste operazioni per la gestione della libreria e dei suoi contenuti, e dall'altra di navigare tra le varie viste e tra gli elementi che le compongono.

Una caratteristica interessante di questo strumento automatico è la possibilità di generare codice C++ a partire dal modello che verrà trattata più nel dettaglio nella sezione A.5.

Infine per ogni vista è prevista la possibilità di validarla, cioè di controllare se il contenuto rispetta le regole imposte dalla metodologia che implementa.

A.2 Supporto alla metodologia di progettazione Coad/Yourdon

Le viste tipiche della metodologia Coad/Yourdon sono le seguenti:

- *modello* (‘*model*’): contiene un diagramma ad oggetti con eventuali raggruppamenti in soggetti;
- *soggetto* (‘*subject*’): contiene l’espansione di un soggetto contenuto in un’altra ‘subject view’ o in una ‘model view’;
- ‘*Object State Diagram*’: diagramma di stato dell’oggetto tipico della metodologia Coad/Yourdon;
- ‘*Service Chart*’: digramma di flusso che descrive il funzionamento di un servizio;
- *voce del dizionario riguradante una Class o un Class&Objects* (‘*Class or Class&Object Dictionary Entry*’): una vista che permette di introdurre informazioni riguardanti una certa classe (astratta o avente istanze);
- *voce del dizionario riguaradante un attributo* (‘*Attribute Dictionary Entry*’): una vista che permette di introdurre informazioni rigurdanti un attributo;
- *voce del dizionario riguradante un servizio* (‘*Attribute Dictionary Entry*’): permette di introdurre informazioni rigurdanti un servizio.

Le viste vengono mantenute all’interno della libreria in modo versionato e si possono modificare solo dopo averle portate in un’apposita zona di lavoro (‘*Workspace*’). La memorizzazione ed il relativo versionamento avvengono al momento del reinserimento nella libreria.

Per realizzare i modelli si può operare direttamente sulla vista grafica, ad esempio inserendo su di essa i nomi dei simboli o le cardinalità delle connessioni. Oppure si possono modificare queste informazioni mediante le *proprietà* degli oggetti che compaiono nelle viste. Le proprietà sono accessibili mediante un apposito comando del menu dell'oggetto che mostra una finestra che permette di inserire le informazioni volute per ognuna delle sue proprietà.

A.3 Descrizione delle classi

La metodologia Coad/Yourdon [CY91a, pag. 156] propone di utilizzare il 'template' riportato in figura A.1 per la definizione di ciascuna classe.

```
specification
   attribute
   attribute
   attribute
   externalInput
   externalOutput
   objectStateDiagram
   additionalConstraints
   notes
   service <name & Service Chart>
   service <name & Service Chart>
   service <name & Service Chart>
```

Figura A.1. Modello per la specifica delle classi.

DECdesign non fornisce un supporto diretto per la compilazione di questo 'template', ma è di ausilio alla realizzazione di quasi tutti gli elementi che lo compongono. Gli elementi non previsti esplicitamente da DECdesign possono essere specificati nelle descrizioni associate ai simboli `Class` e `Class&Objects`.

Le voci `externalInput` ed `externalOutput` servono per specificare i sistemi esterni con cui gli oggetti della classe comunicano. Gli altri elementi del 'template' saranno illustrati nelle sezioni seguenti.

Oltre alle informazioni elencate in figura A.1 si possono aggiungere requisiti sulla tempistica (`timeRequirements`) e requisiti sull'occupazione della

memoria (`memoryRequirements`). Spesso però questi ultimi due tipi di informazioni sono posti all'interno dello strato degli attributi o dei servizi per renderle maggiormente visibili.

I vincoli di esecuzione che riguardano il sistema nella sua globalità sono espressi a parte mediante opportuni documenti che vengono a far parte del prodotto dell'analisi.

Nel seguito è descritta la notazione adottata da DECdesign mettendola a confronto con quella Coad/Yourdon originale.

'Class' e 'Class&Objects'

Una classe è rappresentata mediante il simbolo `Class` che è un rettangolo con angoli arrotondati diviso in tre sezioni (figura A.2). La più alta è dedicata al nome della classe descritta; la parte centrale è dedicata agli attributi mentre quella in basso ai servizi.

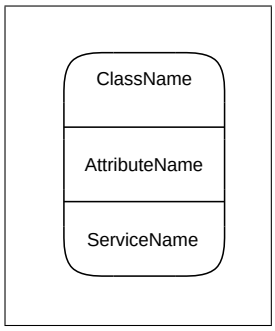

Figura A.2. Simbolo `Class`

Il simbolo `Class` è utilizzato solamente per le classi *astratte*, cioè non aventi alcuna istanza. Il simbolo `Class&Objects` invece rappresenta una classe e tutte le sue istanze. Esso è molto simile al simbolo utilizzato per le classi astratte, ma il rettangolo scuro è circondato da un rettangolo più chiaro che rappresenta gli oggetti che appartengono alla classe in questione. La notazione di DECdesign disegna questo rettangolo esterno con linea tratteggiata ottenendo il simbolo mostrato in figura A.3.

Come ad ogni oggetto nelle 'view' di DECdesign, ad ogni `Class` e `Class&Objects` è associata univocamente una serie di proprietà. Quelle di `Class` e `Class&Objects` sono il nome ed una descrizione.

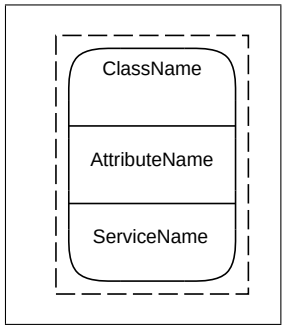

Figura A.3. Simbolo `Class&Objects`

CDE (Class or Class&Objects Dictionary Entry)

Ad ogni `Class` o `Class&Objects` è possibile associare una voce in un apposito dizionario. Ogni voce è costituita da:

- un *nome* che è in genere lo stesso dato alla classe in questione;
- una *descrizione* in cui si possono memorizzare informazioni riguardanti la classe.

Le 'dictionary entry' sono associate alle varie classi in modo non biunivoco. Cioè una stessa voce del dizionario può essere associata a più di un simbolo, mentre un `Class` o un `Class&Objects` può avere al massimo una 'entry' associata. questo fatto le differenzia dalle proprietà dell'oggetto che gli sono invece legate in modo univoco.

Strutture

La notazione adottata da DECdesign coincide perfettamente con quella proposta Coad/Yourdon originale.

La struttura di tipo `Gen-Spec` è descritta da un semicerchio. Una linea collega l'apice dell'arco al simbolo `Class` o `Class&Objects` generalizzazione; i simboli che rappresentano le classi specializzazione sono invece collegati tramite linee che partono dal diametro del semicerchio. Un esemio di struttura `Gen-Spec` è riportato nella figura A.4. Per il collegamento verso il simbolo della classe generalizzazione, invece che una linea semplice, DECdesign utilizza una freccia che punta a tale simbolo.

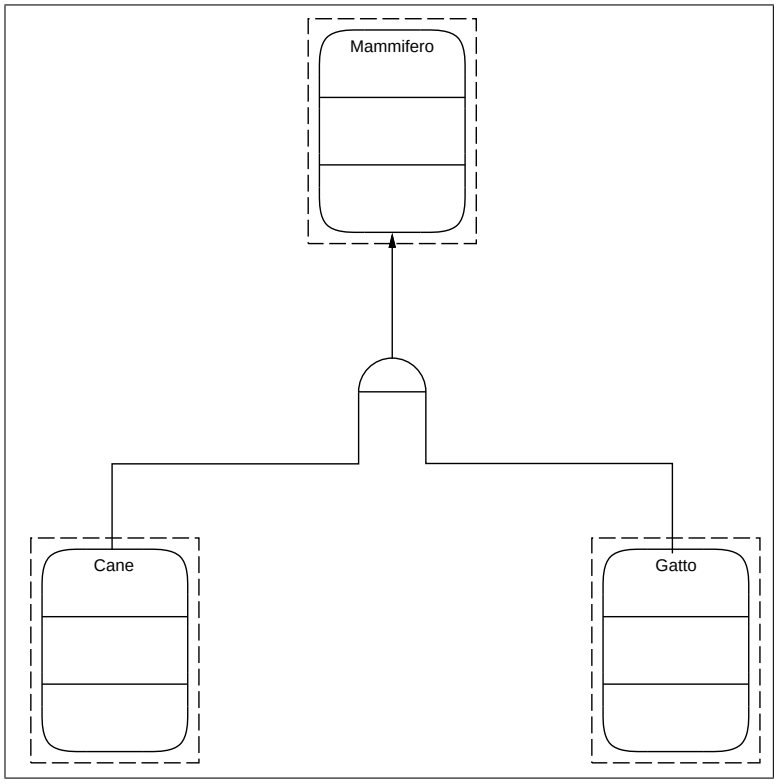

Figura A.4. Utilizzo della struttura `Gen-Spec`

La struttura di tipo `Whole-Part` è rappresentata da un triangolo al cui vertice è collegato tramite una linea la classe contente le istanze composite. A questa connessione è associata una coppia di numeri che indica il minimo ed il massimo numero di parti che costituiscono ciascuno degli oggetti compositi.

Dal centro della base del triangolo parte una linea che collega la classe degli oggetti componenti. Questa connessione ha associata una coppia di numeri che stabiliscono rispettivamente il numero minimo e massimo di oggetti compositi di cui ciascun oggetto parte è componente.

DECdesign nel tracciamento delle connessioni tra i simboli guida l'utente permettendogli di connettere le linee solo in alcuni punti fissi. In particolare per quanto riguarda le strutture, i punti di connessione ammessi sono quelli mostrati nella figura A.6. Si può notare che i simboli `Class&Objects` sono collegati sui lati brevi del rettangolo interno. Questo è corretto per le strutture `Gen-Spec` che rappresentano relazioni a livello classe, ma non per

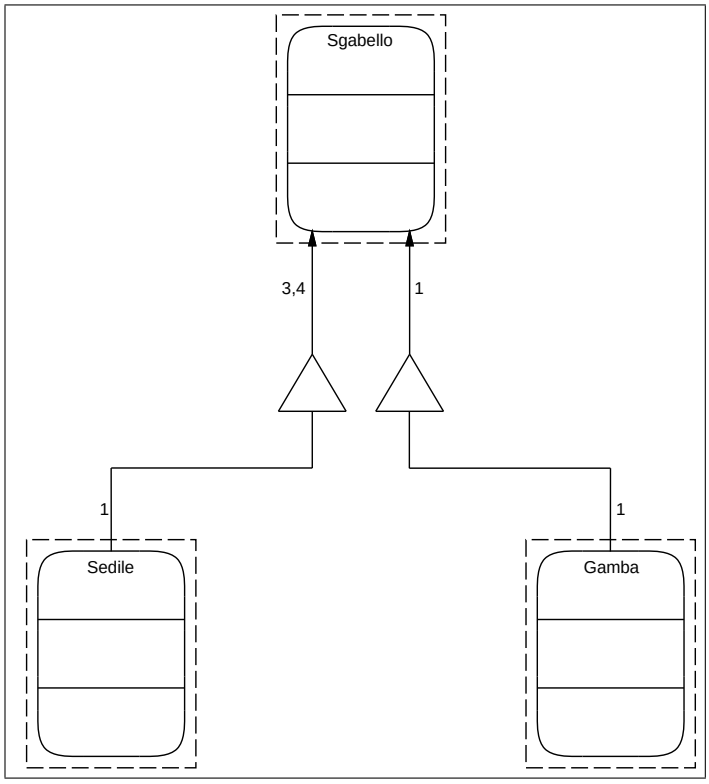

Figura A.5. Utilizzo della struttura `Whole-Part`

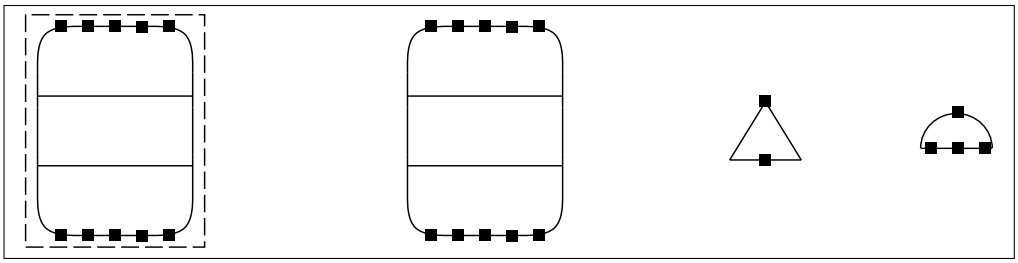

Figura A.6. Punti per le connessioni di struttura

quelle `Whole-Part` i cui estremi dovrebbero essere sul rettangolo esterno che rappresenta gli oggetti della classe. Infatti questa struttura modellizza una relazione tra le istanze delle classi interessate.

La notazione Coad/Yourdon permette di rappresentare l'ereditarietà multipla creando diagrammi a grafo ('lattice') oltre che semplici gerarchie. poiché non è detto che il linguaggio utilizzato supporti l'ereditarietà multipla, quando si passa all'implementazione pò essere necessario rivedere la struttura oppure utilizzare accorgimenti che permettano di mantenere quella prodotta originariamente.

Soggetti

I soggetti hanno tre possibili rappresentazioni:

- *collassata*: un rettangolo contente al suo interno il numero ed il nome del soggetto (figura A.7);

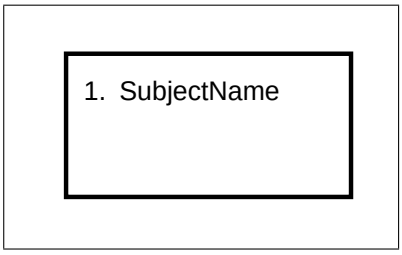

Figura A.7. Soggetto collassato

- *parzialmente espansa*: un rettangolo diviso in due parti: in quella superiore si trovano il nome ed il numero del soggetto, mentre in quella superiore i nomi dei vari `Class` e `Class&Objects` contenuti (figura A.8);

- *espansa*: la metodologia Coad/Yourdon propone di disegnare un rettangolo intorno ai `Class` e `Class&Objects` che contiene; ad ogni angolo si trova il numero del soggetto. DECdesign non utilizza questa notazione, ma semplicemente pone il numero dei soggetti a cui un certo `Class` o `Class&Object` appartengono nella sezione alta del simbolo.

DECdesign permette di associare ad un soggetto una vista a parte. Quindi nelle viste in cui il soggetto è contenuto esso non appare mai in forma espansa. Quando si cerca di espanderlo DECdesign mostra direttamente la 'view' associata. Un problema nasce però dal fatto che non è possibile connettere

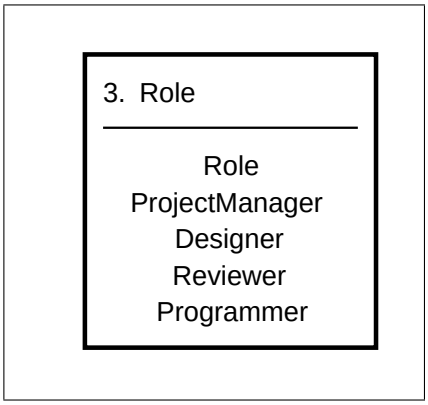

Figura A.8. Soggetto parzialmente espanso

simboli che si trovino in viste diverese, quindi l'associazione di una vista ad un soggetto può essere fatta solo quando gli elementi che esso contiene non siano in alcun modo correlati (connessioni strutturali, di istanza, di servizio) con quelli che si trovano nel resto del modello.

Attributi

Il nome degli attributi è scritto nella sezione centrale del simbolo `Class` o `Class&Objects`. Anche agli attributi sono associate delle proprietà che sono:

- *nome*: lo stesso che appare sul simbolo rappresentante la classe cui appartiene;
- *descrizione*: permette di memorizzare informazioni sull'attributo in questione;
- *tipo del dato* ('*Data-Type*');
- *vincoli*.

ADE (Attribute Dictionary Entry)

Ad ogni attributo si può associare una 'entry' in questo dizionario che permette di memorizzare delle informazioni per le stesse voci elencate per le proprietà degli attributi. Il nome utilizzato nella voce del dizionario può non

essere lo stesso assegnato all'attributo nel modello; infatti la stessa 'entry' può essere condivisa da più attributi.

La differenza di maggior rilievo è però il fatto che le informazioni contenute nella voce del dizionario vengono utilizzate dal generatore automatico di codice C++ di cui si parla nella sezione A.5. Quindi è importante che il tipo dei dati contenuti nell'attributo sia specificato in modo compatibile con la sintassi del liguaggio.

'Instance Connection'

Queste connessioni fanno parte del livello degli attributi ed infatti i punti terminali delle linee che le rappresentano devono essere nella zona centrale dei simboli `Class` e `Class&Objects` (non imposto dalla notazione Coad/Yourdon originale) come mostrato dalla figura A.9.

Ad ogni estremo della connessione si trova una coppia di interi che indicano il numero minimo e massimo di istanze della classe all'altro estremo che sono in relazione con ciascuna istanza della classe che si trova a quell'estremo della connessione.

Infine ad ogni connessione si può associare un'etichetta che permette di evidenziarne le caratteristiche. Questa etichetta, come anche le cardinalià possono essere digitate direttamente sulla vista in cui si trova l' `Instance Connection`, ma anche mediante l'apertura di una 'view' per l'inserimento delle proprietà della connessione.

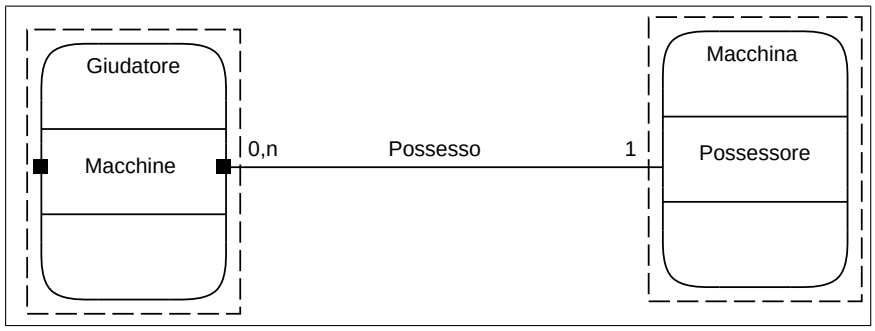

Figura A.9. Punti di connessione per `Instance Connection`

Come si può notare ciascuna `Instance Connection` deve terminare in corrispondenza di un attributo sebbene la metodologia Coad/Yourdon non

richieda nulla del genere. Una tale caratteristica porta a realizzare le connessioni secondo differenti approcci.

- Si inserisce in ogni oggetto un attributo utilizzato per collegarvi tutte le `Instance Connection` di cui necessita la classe che lo contiene. Un possibile nome per questo attributo può essere *id*, cioè un identificatore unico implicito di ciascun oggetto di cui si parla in [CY91a, pag. 123].

- Si realizzano le connessioni sfruttando gli attributi individuati per la classe in questione. Se ci sono classi che necessitano di `Instance Connection` e non hanno attributi a cui collegarle, si aggiunge l'attributo *id*.

- Si realizza ogni connessione introducendo un apposito attributo in ognuna delle classi connesse. Questo procedimento permette di individuare univocamente la connessione mediante gli attributi utilizzati, ma le conseguenze principali sono le seguenti:

  – si può fornire alla connessione una direzione permettendo di assegnare ruoli differenti alle classi connesse;

  – si rispecchia il fatto che per implementare la connessione sarà probabilmente necessario introdurre degli attributi appositi nelle classi insteressate. In realtà questo è un dettaglio implementativo che non ha alcuna importanza a livello di progettazione, ma può aiutare nel passaggio da progetto a realizzazione.

Va infine evidenziato il fatto che secondo la notazione originale della metodologia Coad/Yourdon, i punti terminali delle `Instance Connection` vanno posti sul bordo esterno dei simboli `Class&Objects` perché esse rappresentano dei legami tra gli oggetti e non tra le classi. Invece nella sua implementazione DECdesign attacca le connessioni al rettangolo interno.

Servizi

Il nome dei servizi è riportato nella parte bassa del simbolo `Class` o `Class&Objects`. Ogni servizio ha delle proprietà associate che permettono di specifcare, mediante una opportuna vista di ingresso:

- *nome*;

- *descrizione*;
- *precondizioni*: condizioni che devono essere vere perché il servizio possa essere eseguito;
- *postcondizioni*: devono essere vere alla fine dell'esecuzione del servizio;
- '*External-Input*';
- '*External-Output*': ciò che viene prodotto dall'esecuzione del servizio e probabilmente passato all'oggetto che lo ha invocato;
- *vincoli*.

Queste sono informazioni di tipo descrittivo che non presentano alcuna sintassi particolare. Inoltre non è neanche definita una precisa semantica per esse; ad esempio non è specificato cosa debba succedere se le precondizioni o le postcondizioni non fossero vere, o di che tipo debbano essere i vincoli.

SDE (Service Dictionary Entry)

Come si è detto per gli altri dizionari nelle sezioni precedenti, ad ogni servizio può essere associata una voce nel dizionario in questione ed essa può essere condivisa da più servizi.

Ogni 'entry' permette di specificare le stesse informazioni previste come proprietà del servizio, ma queste vengono utilizzate per generare il codice C++ dei servizi. In realtà il generatore di codice non fa altro che dichiarare per la classe in questione un metodo avente come nome quello specificato, come parametri il contenuto del campo `External-Input` e come valore ritornato ciò che è stato introdotto alla voce `External-Output`. Le altre informazioni introdotte nel dizionario sono riportate nel codice generato semplicemente come commento. Allora è importante, per avere una generazione più efficiente e realmente utilizzabile, riempire i campi `External-Input` ed `External-Output` con espressioni sintatticamente corrette secondo le regole del C++.

‘Service Chart’

Ciò che è fatto dai servizi può essere specificato testualmente nella descrizione, ma la metodologia Coad/Yourdon fornisce un metodo grafico più immediato che è quello delle `Service Chart`. Queste si possono realizzare sfruttando apposite viste che mettono a disposizione una serie di simboli propri di questi diagrammi (figura A.10).

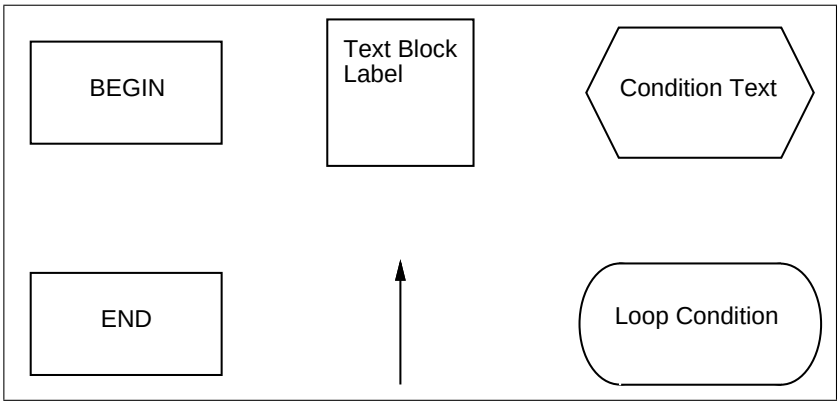

Figura A.10. Blocchi per la realizzazione di `Service Chart`

- *Blocco di inizio* (`Begin`): è posizionato automaticamente in ogni nuova vista e rappresenta il punto di inzio di ciascun servizio. In ogni `Service Chart` si ha un solo blocco di questo tipo perché il punto di ingresso di un servizio è unico.

- *Blocco di fine* (`End`): rappresenta il punto in cui un servizio finisce. Ci possono essere più blocchi di questo genere in ogni `Service Chart`.

- *Blocco di testo* (`Text Block`): serve per esprimere un comportamento che è richiesto al servizio. È rappresentato con un rettangolo contenente al suo interno una descrizione testuale di tale comportamento.

- *Blocco condizionale* (`Condition`): è utilizzato per specificare una condizione in base alla quale intraprendere una certa azione piuttosto che un’altra. è rappresentato da un esagono irregolare contentente al suo interno una descrizione testuale che permette di specificare la condizione discriminante. Dagli angoli formati dai lati brevi partono i due rami

possibili, uno per quando la condizione è vera e l'altro per quando è falsa.

- *Blocco di ciclo* (`Loop`): permette di definire cicli di esecuzione che vengono abbandonati quando la condizione, espressa in modo informale all'interno del simbolo, non risulta soddisfatta. Quindi si devono due rami di esecuzione in ingresso (quello proveniente dai blocchi precedenti e quello che chiude il ciclo) e due rami in uscita (quello che entra nel ciclo e quello che continua l'esecuzione al fallimento della condizione).
- *Connettore* (`Connector`): permettono di unire tra loro i blocchi che definiscono il comportamento del servizio. Il connettore è rappresentato da una freccia che va da un blocco al successivo e permette di definire un ordine tra di essi.

A ciascuna `Service chart` sono associati un nome ed una descrizione.

'Message Connection'

Sono rappresentate mediante frecce che vanno dal mittente al ricevente e, poiché sono gli oggetti che si scambiano i messaggi, i punti estremi di queste frecce dovrebbero trovarsi sul bordo esterno del simbolo `Class&Objects`, ma DECdesign li posiziona sul rettangolo interno.

Figura A.11. Punti di connessione per `Message Connection`

Infatti DECdesign utilizza i punti mostrati in figura A.11 in corrispondenza dei nomi dei servizi delle classi connesse. Questo fatto permette di identificare in quale servizio del mittente avviene l'invocazione del messaggio

ed quale servizio del ricevente viene eseguito a seguito del messaggio rappresentato dalla `Message Connection`.

La metodologia Coad/Yourdon però non prevede un tale significato per le `Message Connection`, per cui per tracciarne di più generali si possono utilizzare stratagemmi simili a quelli presentati nella sezione A.3 per le `Instance Connection`.

Ad ogni `Message Connection` si può associare un'etichetta che la identifica; di valido aiuto sarebbe anche la possibilità di tracciare con colori differenti le connessioni che identificano scambi di messaggi dovuti a processi concorrenti distinti.

'Object State Diagram'

Possono essere costruiti utilizzando apposite 'view' che mettono a disposizione alcuni blocchi fondamentali mostrati dalla figura A.12.

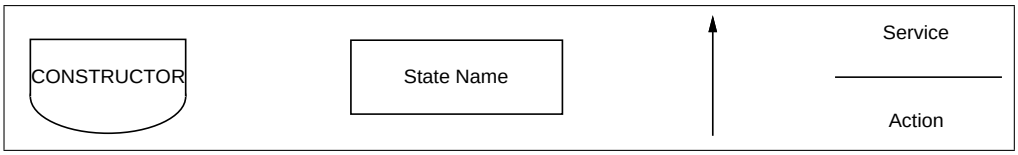

Figura A.12. Vista per la realizzazione di `OSD`

- *Stato iniziale* (`Contructor`): stato iniziale in cui si trova l'oggetto nel momento in cui viene istanziato. Ogni diagramma di stato ne possiede solamente uno che è inserito automaticamente quando si comincia a costruire un nuovo diagramma di stato.
- *Stato* (`State`): si tratta di un rettangolo con un'etichetta al suo interno che serve ad identificarlo. Generalmente uno stato è identificato dal valore delle variabili che lo caratterizzano.
- *Transizione* (`Transition`): è un arco orientato che collega due stati successivi stando ad indicare che è possibile che l'oggetto passi dallo stato in cui ha origine la freccia a quello da essa puntato.
- *Servizio/Azione* (`Service/Action`): si tratta di una linea orizzontale con due etichette; questo simbolo deve sempre essere associato ad

una transizione. L'etichetta superiore indica il nome del servizio che causa il cambiamento di stato, mentre quella inferiore descrive l'azione che accompagna il cambiamento di stato (generalmente l'invio di un messaggio).

A.4 Selezione degli strati

Uno degli aspetti principali della metodologia Coad/Yourdon è la visione del modello in strati ('layers') che offrono varie visioni del sistema che consentono di limitare la quantità di informazioni sottoposta all'osservatore.

È possibile imporre a DECdesign di mostrare i modelli in modo completo, cioè visualizzando gli elementi di tutti gli strati, oppure solo quelli relativi agli strati specificati.

La suddivisione in strati riveste un'importanza pratica nella realizzazione e comprensione dei modelli. Infatti nell'esaminare un modello ad oggetti può essere di aiuto in un primo momento focalizzare l'attenzione su quali sono le classi, quindi sulle relazioni strutturali tra esse e solo alla fine scendere ad analizzare attributi e servizi. Inoltre gli strati degli attributi e dei servizi comprendono anche le connessioni che, anche in modelli di medie dimensioni, possono essere in numero elevato e creare un intreccio di linee che complica la visione del progetto.

A.5 Principali vantaggi e problemi riscontrati

I vantaggi nell'uso dei DECdesign si sentono particolarmente nella possibiltà di muoversi da un documento all'altro del modello con estrema facilità. Cioè, selezionato un `Class&Objects` all'interno di una vista di modello o di soggetto, è immediato accedere alla vista del suo `Object State Diagram` oppure arrivare al modello cui appartiene un oggetto di cui si sta esaminando la 'entry' nel corrispondente dizionario. DECdesign presenta però alcuni problemi di carattere puramente implementativo che ne rendono scomodo l'utilizzo.

Dimensioni delle viste Come già accennato in precedenza, non è possibile connettere `Class&Objects` che si trovino in viste diverse. Ciò implica innanzitutto che non è possibile associare ad un'altra vista l'espansione di un

soggetto che contenga simboli da collegare ad altri all'esterno. Questo fatto porta come conseguenza che un progetto rischia di dover essere realizzato completamente su una stessa 'model view' che assume dimensioni notevoli. Una tale vista diventa così molto difficile da consultare perché gli elementi che la compongono si trovano fisicamente posti a notevole distanza. Cone immediata conseguenza le connessioni tra essi divengono molto lunghe, e quindi difficili da seguire.

Aumentando le dimensioni della vista, aumenta anche il numero di simboli in essa contenuti e quindi lo spostamento della zona visualizzata sullo schermo diviene sempre più lento. Inoltre per evitare che le connessioni passino sopra ad altri simboli rendendo più defficoltosa la lettura del modello, occorre realizzare percorsi sempre più tortuosi e difficili da seguire durante l'esame del progetto.

Questi problemi non si possono aggirare collassando i soggetti e amntenendone solo uno alla volta espanso. Infatti quando si collassa un oggetto, la sua icona è piccola, ma la dimensione della vista non può essere ridotta al di sotto dell'estensione occupata dal soggetto nella forma espansa.

Formati di stampa Una difficoltà di tipo pratico nasce anche dal tipo di stampe generate dallo strumento di queste viste. Infatti sono previsti solamente due formati: uno secondo cui tutto il diagramma è scalato in modo da essere contenuto in un singolo foglio, un altro a dimensione fissa per cui il progetto viene diviso su vari fogli. Per un modello piuttosto semplice come quello dell'esempio del processo, il secondo formato di stampa ha richiesto 68 folgi organizzati su 4 righe e 17 colonne. È evidente che un tale formato di stampa risulta illeggibile come anche quello su singola pagina che è invece troppo piccolo.

Inoltre DECdesign permette di generare dei rapporti ('report') del progetto in cui dovrebbero essere raccolti tutti i diagrammi e le informazioni sulle entità definite. In realtà, sebbene in questi rapporti siano inclusi tutti i diagrammi (non sempre in un formato leggibile), non vengono incluse molte delle informazioni introdotte, come ad esempio le proprietà degli oggetti presenti nelle viste, oppure le voci dei vari dizionari. Diviene quindi un lavoro di enormi dimensioni andare a stampare tutte le informazioni riguardanti un progetto accedendo ad esse separatamente attraverso le rispettive viste.

Tracciamento delle connessioni Le connessioni all'interno delle viste, possono essere tracciate in modo completamente automatico selezionando semplicemente gli elementi da collegare. oppure, per poter evitare che queste si sovrappongano agli elementi presenti nel modello, si ha la possibilità di specificarne manualmente il percorso ('*route*').

Quando però un soggetto viene collassato tutte le connessioni che sono legate ad elementi contenuti in esso, vengono spostate per portarle tutte quante a puntare all'icona del soggetto. quando lo si espande nuovamente le linee sono ritracciate verso gli elemtni collegate, ma non sono più seguiti i percorsi definiti dall'utente causando una notevole confusione.

Lo stesso problema si riscontra quando, dopo aver imposto di non visualizzare un 'layer', lo si rende nuovamente visibile. Le connessioni di questo strato sono allora tracciate nello stesso modo descritto nel caso precedente.

Questo fatto rende inutilizzabile sia il collassamento dei soggetti che la possibilità di nascondere alcuni strati. Infatti per poterli utilizzare occorre ogni volta ridefinire tutti i percorsi dei collegamenti.

'Editor' grafico dei modelli e dei soggetti Ha alcune limitazioni che ne rendono pesante l'uso. In particolare non è possibile copiare parti del modello, oppure singole icone, per riprodurle altrove.

È possible spostare i singoli elementi, ma non una intera porzione del siagramma. Quindi, volendo spostare un gruppo di `Class&Objects` collegati da certe connessioni, si devono spostare ad uno ad uno i simboli interessati e ricostruire gli opportuni percorsi per le connessioni.

Di valido aiuto sarebbe anche la possibilità di poter spostare oggetti da una vista all'altra permettendo di riutilizzarli in altri progetti o anche solo di muoverli all'interno dello stesso. La mancanza di questa caratteristica richiede di dover cominciare ogni nuovo progetto dal nulla senza poter utilizzare parti di altri. poiché l'unico tipo di riutilizzo possibile è quello di intere viste, si può parzialmente aggirare l'ostacolo importando 'view' da altri progetti e cancellandone le parti che non interessano. Resta impossibile comunque integrare differenti viste in una unica.

Generatore di codice La generazione del codice può essere fatta globalmente per l'intero modello, per le singole viste, o anche per i singoli `Class&Objects`. Il codice generato si limita alla dichiarazione delle classi

con la loro gerarchia di ereditarietà, i nomi degli attributi e dei servizi. Questi sono accompagnatio da tutta una serie di commenti che riportano le varie informazioni introdotte per gli elementi componenti il modello.

La povertà del codice generato è ovviamente imputabile al fatto che la informazioni specificate per le classi non sono date in modo formale o con una semantica ben precisa, per cui non se può ricavare codice dettagliato.

Anche i diagrammi di stato degli oggetti e le `Service Chart` non sono utilizzate in alcun modo, perché realizzati in modo troppo informale.

Infine nel codice generato non sono considerate in alcun modo le strutture `Whole-Part`, le `Instance Connection` e le `Message Connection`.

Appendice B

Smalltalk-80

Per poter descrivere l'implementazione di S^3 è stato necessario fornire alcune informazioni sul linguaggio ad oggetti Smalltalk-80 e sull'ambiente di programmazione Objectworks(r)/Smalltalk. In questa appendice, dopo avere elencato le carateristiche di maggior rilievo di Smalltalk, sono stati spiegati gli elementi fondamentali del linguaggio e l'organizzazione di Objectworks(r)/ Smalltalk. Segue qundi una descrizioni delle principali classi predefinite che sono state utilizzate per la realizzazione di S^3, in particolare per la gestione delle interfacce utente e della concorrenza.

B.1 Caratteristiche principali

Per l'implementazione di S^3 si è utilizzato Objectworks(r)/Smalltalk [Sys90], Release 4 per SunOS 4.1.1, che è un linguaggio orientato agli oggetti ('Object Oriented Programming Language') largamente utilizzato nella realizzazione di prototipi grazie ai supporti che fornisce per la scrittura e la correzzione ('debugging') dei programmi e grazie alla totale portabilità del codice da esso generato. Non è molto utilizzato nella produzione di software commerciale.

B.1.1 Principali punti di forza

Mettiamo in primo luogo in evidenza una serie di vantaggi offerti da Objectworks(r)/Smalltalk, in quanto linguaggio di programmazione, che ne accrescono la potenza e ne facilitano l'uso, sia nella produzione di normali applicativi, che nella modellizzazione dei processi.

Uniformità Qualsiasi meccanismo e costrutto in Smalltalk è implementato mediante classi, oggetti ed invocazione dei loro metodi. Un esempio significativo è dato dalla struttura della selezione condizionale che è presente in qualsiasi linguaggio di programmazione. In Smalltalk essa è implementata mediante l'invocazione di un apposito metodo degli oggetti della classe `Boolean`, di cui è istanza la condizione che regola la selezione, passando come parametro il codice da eseguire nel caso le condizione sia vera e nel caso questa sia falsa. Un programma in esecuzione non è altro che un insieme di istanze delle classi definite in Smalltalk che invocano le une i metodi delle altre creando eventualmente nuovi oggetti.

Completa portabilità Objectworks(r)/Smalltalk è costituito da un insieme di classi predefinite la cui implementazione è compilata originando un 'file' detto *immagine* che viene eseguito da una macchina virtuale. Per ogni piattaforma hardware esiste una diversa macchina virtuale, ma il codice immagine che queste eseguono è identico per tutte. Allora per eseguire uno stesso programma su diverse piattaforme hardware, è sufficiente trasportare su ognuna il 'file' immagine del programma ottenuto dalla compilazione delle classi predefinite e di quelle aggiunte dal programmatore.

Programmazione interattiva Objectworks(r)/Smalltalk mette a disposizione una serie di interfacce per accedere alle definizioni delle classi presenti, modificarle ed eventualmente aggiungerne altre. Sono fornite inoltre facilitazioni per navigare le strutture di ereditarietà che collegano le classi, per raggiungere le classi che implementano un certo metodo o le cui istanze invocano un certo metodo.

Creazione interattiva delle istanze È possibile creare in modo interattivo istanze delle varie classi presenti nel sistema e mandare messaggi alle istanze create. L'esecuzione di un programma comincia proprio creando uno o più oggetti che a loro volta provvederanno ad istanziarne altri ed a comunicare mediante l'invocazione reciproca dei metodi.

Compilazione dinamica dei metodi Le definizioni delle classi ed il codice dei loro metodi sono compilati separatamente ed in modo dinamico; cioè quando si cambia il codice di un metodo è sufficiente ricompilare il singolo metodo e non è necessario ricompilare tutto l'insieme delle classi o anche

solo l'intera classe che contiene il metodo in questione. Questa caratteristica è molto importante sia per la correzzione dei programmi che per la modifica di questi ultimi che può essere fatta senza interromperne l'esecuzione. Notevole importanza è rivestita da questa caratteristica nel momento in cui si utilizzi Smalltalk per l'implementazione di modelli di processo, perché ne permette l'evoluzione, cioè l'apporto di modifiche senza doverne interrompere l'attuazione.

Rappresentazione esplicita delle classi Objectworks(r)/Smalltalk prevede una struttura di meta-classi che forniscono una rappresentazione esplicita ed accessibile delle classi presenti nel sistema. Questo conferisce *riflessività* al linguaggio che è una caratteristica fondamentale dei linguaggi per la modellizzazione dei processi in modo da garantire ai modelli la possibilità di evolvere.

Supporto all'individuazione e correzione degli errori Una delle fasi più importanti e delicate della programmazione, è l'individuazione degli errori, sempre presenti in prodotti di qualsiasi dimensione. Objectworks(r)/ Smalltalk fornisce un valido supporto per individuare la causa degli errori procedendo passo a passo e seguendo le varie invocazioni di messaggi da un oggetto all'altro. una volta individuato un errore è possibile correggerlo e riprendere l'esecuzione da dove era stata interrotta. Si ha anche la possibilità di inserire nel codice dei metodi dei punti di interruzione ('*breakpoint*') in cui l'esecuzione viene sospesa per passare il controllo a questi strumenti di supporto al '*debugging*'.

B.1.2 Caratteristiche relative all'orientamento agli oggetti

poiché Smalltalk è un linguaggio 'object oriented', esso prevede concetti tipici del modello ad oggetti quali quelli di classe, ereditarietà ed istanza.

Gerarchia delle classi In Objectworks(r)/Smalltalk qualsiasi classe ha una superclasse da cui eredita tutti gli attributi ed i metodi; l'unica classe che non deriva da nessun'altra è `Object` che è antenata di tutte le altre classi. Quando si crea una nuova classe, se questa non è specializzazione di nessun altra, in particolare le si assegna come superclasse `Object`.

Ad ogni classe è associata una *meta-classe*: questa è una classe definita come qualsiasi altra, ma ha una sola istanza che è la rappresentazione della classe corrispondente. Se la classe `C1` è generalizzazione di `C2`, allora la metaclasse `M1` di `C1` è generalizzazione della metaclasse `M2` di `C2`. Quindi, oltre alla gerarchia delle classi, si ha una gerarchia parallela, cioè che presenta la stessa struttura, delle meta-classi.

Ereditarietà Objectworks(r)/Smalltalk supporta solamente l'ereditarietà singola ('single inheritance'), ma questo non rappresenta un grosso limite, sia perché il progetto ad oggetti che si vuole implementare non ne fa uso, sia perché esistono metodi, specifici per Smalltalk o generali, per realizzare strutture basate sull'ereditarietà multipla mediante un linguaggio che supporti solamente quella singola.

Ogni sottoclasse eredita dalla propria generalizzazione tutti gli attributi e tutti metodi. La sottoclasse può quindi dichiararne altri, o anche ridefinire l'implementazione dei metodi, ma non è possibile ridefinire gli attributi. Questo d'altronde non è indispensabile dal momento che Smalltalk non supporta la tipizzazione ('typing') per cui il valore assegnato agli attributi degli oggetti specializzazione non devono essere dello stesso tipo di quelli assegnati agli attributi degli oggetti generalizzazione.

Accessibilità degli attributi Gli attributi di un oggetto, detti `instance variable` in Smalltalk, non sono in alcun modo accessibili da parte di altri oggetti, neanche in lettura, per cui qualsiasi operazione su di essi deve essere fatta mediante l'invocazione dei metodi messi a disposizione dalla classe. Inoltre un attributo non è altro che un identificatore che permette di individuare un oggetto che rappresenta il valore dell'attributo stesso; questo è in linea con l'uniformità che caratterizza Smalltalk, perché tutto è basato su classi, istanze ed invocazioni dei metodi di queste.

B.2 Espressioni letterali

In Smalltalk possono apparire cinque tipi di espressioni letterali che sono anch'essi istanze di ben precise classi [GR83, pag. 19].

1. **Numeri:** sono oggetti che rappresentano valori numerici e rispondono a messaggi che permettono di calcolare risultati matematici. La

rappresentazione letterale dei numeri è una sequenza di cifre, che può essere preceduta da un segno meno e seguita da un punto decimale, ed un'altra sequenza di cifre. Si possono avere rappresentazioni in basi diverse. Esempi:

```
456847
-25.4
```

2. **Caratteri:** rappresentano un singolo simbolo di un alfabeto e la loro rappresentazione letterale consta in un segno `$` seguito da un qualsiasi carattere. Esempi:

```
$a
$$
```

3. **Stringhe:** sequenze di caratteri che rispondono a messaggi per accedere ai singoli caratteri, fare concatenazioni e le altre tipiche operazioni eseguibili sulle stringhe. La rappresentazione è una sequenza di carateri contenuta all'interno di apici singoli. Esempi:

```
'stringa'
'Process Modeling'
```

4. **Simboli:** sono oggetti utilizzati come nomi all'interno del sistema e possono essere usati come valori costanti su cui fare eventualmente confronti. La rappresentazione letterale è una sequenza di caratteri preceduta dal simbolo `#`. Esempi:

```
#simbolo
#completed
```

5. **Vettori:** sono una semplice struttura dati il cui contenuto può essere acceduto mediante un indice intero il cui valore varia tra uno ed il numero totale di elementi contenuti. La rappresentazione è una sequenza di altre espressioni letterali separate da spazi e racchiuse all'interno di parentesi tonde, il tutto preceduto dal simbolo `#`. Esempi:

```
#(1 2 3)
#('una stringa' #un_simbolo (15 $R) 128)
```

B.3 Le variabili

Smalltalk non presenta modularità, per cui gli oggetti che sono creati in qualsiasi modo (interattivamente dall'utente o dall'esecuzione di un metodo da parte di un altro oggetto), sono visibili a qualsiasi oggetto presente nel sistema. L'unico vincolo per poter accedere ad un oggetto è rappresentato dalla necessità di disporre di un riferimento a tale oggetto; questo compito viene assolto dalle variabili.

Alle variabili possono essere assegnati dei valori, cioè si può imporre che individuino un oggetto ed attraverso di esse sarà possibile accedere agli oggetti in questione, ovvero invocarne i metodi.

Quando un oggetto è creato esso è memorizzato in una zona di memoria detta *heap* ed è accessibile mediante una o più variabili che lo identificano [GR83]. L'oggetto rimane nella 'heap' fino a che c'è almeno una variabile che gli fa riferimento.

B.3.1 Visibilità

Una variabile può essere dichiarata in modi differenti e di conseguenza essa acquista un diverso ambito di visibilità.

- Dichiarando una variabile come *condivisa* ('*shared variable*'), essa può essere accessibile da più oggetti; questo è ottenuto inserendola in un insieme, detto '*pool*', per cui è definito il campo di visibilità voluto. Per convenzione il nome delle variabili condivise viene scritto con la prima lettera maiuscola. Si possono individuare due '*pool*' di sistema:

  1. `Smalltalk` continene tutte le variabili globali, cioè visibili da parte di qualsiasi oggetto;
  2. uno per ogni classe che contiene gli attributi di livello classe ('*class variable*') i cui elementi sono visibili alla classe ed a tutte le sue istanze.

- Dichiarando una variabile come *attributo* di un oggetto ('*instance variable*'), essa è visibile solamente all'oggetto stesso.

- Dichiarando una variabile *temporanea* all'interno di un metodo ('*temporary variable*'), essa è visibile solamente all'interno del corpo del metodo ed ha un oggetto associato solo fino a che tale metodo è in esecuzione.

Un particolare esempio di variabile globale è dato dal nome di una qualsiasi classe. Infatti le classi non sono altro che oggetti individuati mediante una variabile che è visibile da qualsiasi altro oggetto presente nel sistema il cui nome coincide con quello della classe.

B.3.2 Pseudo-variabili

In Smalltalk esistono alcune pseudo-variabili cui sono associati, in modo fisso o dinamico, ben determinati oggetti, per cui ad esse non è possibile fare assegnazioni.

nil fa riferimento ad un oggetto che viene assegnato ad una variabile quando nessun altro oggetto è appropriato. Quando una variabile viene creata, prima di essere inizializzata, punta a questo oggetto.

true fa riferimento ad un oggetto che rappresenta il valore booleano 'vero'.

false fa riferimento ad un oggetto che rappresenta il valore bolleano 'falso' [GR83, pag. 23].

self ha senso solo se utilizzata nell'implementazione dei metodi e fa riferimento all'oggetto che sta eseguendo il metodo, cioè quello che ne ha ricevuto l'invocazione [GR83, pag. 50].

super anche questa variabile ha senso solo quando è utilizzata all'interno dell'implementazione di un metodo ed individua l'oggetto che sta eseguendo il metodo, ma la selezione dei metodi che sono invocati mediante questa variabile è realizzata in modo particolare [GR83, pag. 63] (vedi sezione B.4.2).

B.4 I metodi

I metodi sono memorizzati nella classe in cui sono definiti e sono disponibili a tutte le istanze di tale classe. Grazie all'uniformità di Smalltalk, che vede le classi come oggetti qualsiasi, è possibile anche definire dei metodi di livello classe che possono essere invocati sulle classi; uno dei compiti più frequentemente assegnati a questi metodi è la creazione delle istanze.

B.4.1 Identificazione dei metodi

In Smalltalk l'interazione tra gli oggetti avviene mediante l'uso di messaggi. Un messaggio individua un ricevente, un *selettore* che permette di identificare il metodo corrispondente ed eventualmente un certo numero di argomenti che sono passati a tale metodo. Ci sono diversi tipi di messaggi catalogati in base al tipo di selettori ed al numero argomenti che richiedono.

Un grosso vantaggio dell notazione proposta per la scrittura dei messaggi, è che se si scelgono con cura i nomi delle variabili ed i selettori, permette di conferire un'elevata leggibilità al sorgente dei metodi.

Messaggi unari

Sono caratterizzati da un singolo selettore che permette, da solo, di individuare il metodo da eseguire; non prevedono il passaggio di alcun argomento. L'invocazione del metodo identificato dal selettore `selettore` sull'oggetto individuato dalla variabile `destinatario` si ottiene mediante il messaggio:

```
destinatario selettore
```

Messaggi con parole chiave

Rappresentano il tipo più generale di messaggio che individua un metodo a cui vengono passati uno o più argomenti. Un mesaggio di questo tipo ('*keyword message*') è costituito dalla variabile che individua il destinatario seguita da una o più '*keyword*', ciascuna delle quali precede un argomento [GR83, pag. 26].

```
destinatario keyword1:  argomento1 keyword2:  argomento2
...
```

Il selettore del messaggio è costituito dalla concatenazione delle varie parole chiave.

Messaggi binari

È un tipo di messaggio che ha un singolo argomento (che segue il selettore), ma che presenta una notazione differente rispetto a quella offerta dai '*keyword message*' aventi una sola parola chiave. Il selettore del messaggio è costituito da uno o due caratteri non alfanumerici, con la restrizione che il secondo

non sia un segno meno [GR83, pag. 27]. I selettori binari sono utilizzati per lo più per mesaggi di tipo aritmetico in modo da poter mantenere la stessa notazine utilizzata nellamatematica e nei linguaggi tradizionali. Esempi di messaggi binari sono:

```
3 + 5
part <= total
```

B.4.2 Meccanismo di selezione dei metodi

Come evidenziato trattando dell'ereditarietà in Smalltalk-80, i metodi definiti per una superclasse sono ereditati dalle sue sottoclassi, che li possono comunque ridefinire. La possibilità di invocare il metodo da parte delle sottoclassi non implica che il codice sia ripetuto in esse, ma è garantita dal meccanismo utilizzato da Smalltalk per selezionare il metodo corrispondente ad un messaggio. Questo fornisce anche il vantaggio che, quando un metodo è ridefinito, può comunque al suo interno invocare l'esecuzione del corrispondente metodo implementato nella generalizzazione.

Quando Smalltalk si trova a dover eseguire l'invio di un messaggio compie i passi elencati nel seguito:

1. ricava l'oggetto destinatario identificato dalla variabile cui è associato il selettore;
2. risale alla classe di tale oggetto;
3. mediante il selettore controlla se la classe in questione dispone del metodo corrispondente:
   - se tale metodo è presente lo esegue;
   - se non è presente risale la gerarchia di erediarietà della classe in questione e ad ogni passo controlla se il metodo voluto è stato definito nella classe attualmente considerata.
4. s Se nel risalire la gerarchia delle generalizzazioni giunge alla classe `Object`, e anche questa non fornisce un'implementazione per il metodo, viene dato un messaggio di errore.

Non sempre l'identificazione dell'oggetto cui è destinato il messaggio è banale, in particolare quando vengono utilizzate le pseudo-variabili `self` o `super`. Per capire come queste vengano trattate va tenuto presente che qualsiasi invio di messaggio fa parte dell'esecuzione di un certo metodo.

- `self` si riferisce all'oggetto che sta eseguendo il metodo che manda il messaggio, anche qualora questo metodo non sia definito nella classe di tale oggetto, ma in una superclasse.
- `super` si riferisce alla superclasse della classe in cui è definito il metodo che sta eseguendo l'invio del messaggio, e non alla superclasse della classe dell'oggetto che sta eseguendo tale metodo.

B.5 Organizzazione delle classi

Le classi che sono fornite da Objectworks(r)/Smalltalk e quelle che costituiscono le applicazioni sviluppate utilizzandolo, sono raggruppate in *categorie* ('*class category*'), a seconda del loro impiego o delle generalizzazioni da cui derivano. Questa suddivisione rende più semplice individuare le classi per poterle modificare o anche semplicemente visionare mediante le apposite interfacce fornite dal sistema.

Per ogni classe è possibile accedere alla sua definizione ed all'implementazione dei metodi che sono suddivisi tra quelli di livello istanza e quelli di livello classe. Inoltre i metodi di uno stesso livello sono ancora raggruppati in *protocolli* in base al tipo di servizio che forniscono. Questo raggruppamento, analogamente a quello delle classi in categorie, permette di individuare più facilmente i metodi per modificarne o vederne l'implementazione.

B.5.1 La definizione delle classi

La definizione di una classe è fatta seguendo il modello, fornito da Objectworks(r)/Smalltalk, riportato nella figura B.1. È necessario specificare la superclasse da cui la classe che si vuole creare deriva per ereditarietà, quindi il nome della classe. È possibile, ma non indispensabile, fornire l'elenco degli attributi di livello istanza e di livello classe, ed eventualmente i 'pool' su cui gli oggetti della classe avranno visibilità. Infine si deve specificare la categoria a cui la nuova classe appartiene.

```
NameOfSuperclass subclass:  #NameOfClass
   instanceVariableNames:  'instVarName1 instVarName2'
   classVariableNames:  'ClassVarName1 ClassVarName2'
   poolDictionaries:  ''
   category:  'NameOfCategory'
```

Figura B.1. Modello per la definizione delle classi

B.5.2 L'implementazione dei metodi

Anche per la definizione dei metodi di una classe Objectworks(r)/Smalltalk fornisce un modello ed è riportato nella figura B.2. Sulla prima riga si scrive il selettore del metodo, eventualmente formato da un certo numero di 'keyword' ognuna seguita da un argomento.

```
message selector and argument names
   "comment stating purpose of message"

   | temporary variable names |
   statements
```

Figura B.2. Modello per la definizione dei metodi

Si può includere, racchiudendolo tra doppi apici, un commento che spiega il funzionamento del metodo ed i servizi che offre. Questo riveste una notevole importanza in un ambiente come quello offerto da Objectworks(r)/Smalltalk perché, dal momento che le definizioni delle varie classi e dei loro metodi sono interattivamente disponiblili al programmatore, questi commenti possono fornire importanti informazioni sull'uso delle classi e dei loro metodi divenendo un'efficiente documentazione 'on line' del sistema.

Si ha la possibilità di definire, racchiudendole tra una coppia di barre verticali (|), variabili temporanee la cui vita e visibilità sono limitate all'esecuzione del metodo. A queste segue l'elenco dei comandi che costituiscono il corpo del metodo; ognuno di questi è un messaggio che è separato dagli altri da un punto.

Ogni metodo deve restituire un valore, cioè in pratica un oggetto, che può essere specificato utilizzando l'operatore ↑. Un comando del tipo

```
↑variable
```

causa l'interruzione dell'esecuzione del metodo in cui si trova ed il ritorno al punto in cui era avvenuta l'invocazione, restituendo l'oggetto identificato da `variable`. Se durante l'esecuzione di un metodo non è raggiunto un comando di questo genere, al termine viene automaticamente restituito l'identificatore dell'oggetto che ha eseguito il metodo, cioè come se fosse stato eseguito il comando:

```
↑self
```

B.6 Principali classi fornite da Objectworks(r)/Smalltalk utilizzate per S^3

Objectworks(r)/Smalltalk mette a disposizione del programmatore un grande numero di classi che possono essere utillizzate nella scrittura di applicazioni, sia creando nuove classi a partire da esse, sia istanziando i loro oggetti su cui invocare i metodi. Alcune di queste classi sono brevemente descritte nel seguito perché sono molto generali e possono rivelarsi particolarmente utili in molti casi, ed in particolare sono state largamente impiegate nella realizzazione di S^3.

B.6.1 Blocchi

Un blocco è un'istanza della classe `BlockClosure` ed è costituito da un insieme di comandi racchiusi tra parentesi quadre; in un blocco è possibile anche definire variabili temporanee la cui visibilità è limitata al blocco stesso. La forma più generale per un blocco è riportata in figura B.3.

```
[| temporary variable names |
statement1.
...
statementN]
```

Figura B.3. Forma generale di blocco

Come qualsiasi altro oggetto, i blocchi possono essere passati come argomento delle invocazioni a metodo, permettendo ad esempio, di realizzare

strutture di controllo del flusso come i cicli di ripetizione condizionale o la selezione condizionale. Inoltre, come avviene per qualsiasi altro oggetto, è possibile invocare dei metodi sui blocchi che ad esempio ne causano l'esecuzione con la possibilità di passare eventualmente dei parametri.

Un esempio dell'uso dei blocchi è la realizzazione della ripetizione condizionale:

```
block1 whileTrue:  block2
```

L'invio di questo messaggio implica l'invocazione del metodo `whileTrue:` sull'oggetto `block1` passando come argomento l'oggetto `block2`. L'esecuzione del metodo causa l'esecuzione dei comandi contenuti in `block1`. Come quasi sempre accade in Smalltalk, ogni comando corrisponde all'invocazione di un metodo che ritorna un valore. Se il valore restituito dall'ultimo messaggio contenuto in `block1` corrisponde al valore booleano vero, viene eseguito `block2` e quindi nuovamente `block1`, realizzando appunto un ciclo di ripetizioni condizionali.

B.6.2 `Array`

Si tratta di una delle classi di maggiore utilità nello sviluppo di applicazioni in Smalltalk ed è utilizzata per realizzare vettori. La caratteristica particolare rispetto agli altri linguaggi di programmazione, è che gli elementi contenuti in un vettore possono essere oggetti di qualsiasi classe, e non necessariamente tutti della stessa.

Le istanze possono essere create con il metodo `new:`, che istanzia un oggetto che ha tante componenti quante sono specificate dall'argomento e che ha tutti gli elementi inizializzati a `nil`. Oppure mediante il messaggio `with:with:...` che dopo ogni parola chiave ha come argomento un oggetto da inserire nel vettore con cui viene inizializzata la corrispondente componente.

L'inserimento di elementi può essere fatto mediante l'invio del messaggio

```
anArray at:  position put:  item
```

che inserisce l'elemento `item` alla posizione `position` nel vettore `anArray`; la dimensione di `anArray` deve essere maggiore o uguale a `position`.

L'elemento che è memorizzato in una certa posizione può essere ottenuto come valore di ritorno dell'invocazione al metodo `at:`.

B.6.3 `Set`

Le istanze di questa classe sono insiemi in cui può essere contenuto un numero qualsiasi di oggetti di qualunque classe. I metodi invocabili sulle istanze di `Set` permettono di inserire nuovi elementi nell'insieme, eliminare elementi, scorrere ad uno ad gli elementi contenuti eseguendo delle operazioni su di essi. Sono possibili anche elaborazioni sulle istanze di `Set` che implementano le classiche operazioni tra insiemi.

La classe `Set` fornisce al programmatore una struttura molto potente e versatile per raggruppare oggetti in modo piuttosto libero, con l'unico vincolo di non poter inserire in un insieme due volte la stessa istanza. Questo è ottenibile utilizzando oggetti della classe `Bag` invece che `Set`.

Volendo invece raggruppamenti con caratteristiche particolari, ad esempio i cui elementi sono ordinati, si può ricorrere ad classi create come specializzazione delle suddette e già fornite da Objectworks(r)/Smalltalk.

B.6.4 `Dictionary`

È una specializzazione della classe `Set` e le sue istanze sono caratterizzate dal fatto che ogni elemento è costituito da una *chiave* ('*key*') che lo identifica univocamente, ed un *valore* ('*value*'). Questo rende possibile accedere ad un ben preciso valore contenuto in un oggetto `Dictionary` utilizzando la chiave corrispondente.

Questa classe è stata largamente utilizzata nella realizzazione di S^3 e costituisce uno dei tipi di diato astratto più efficienti e potenti tra quelli messi a disposizione da Smalltalk.

B.7 L'interfaccia utente

Smalltalk fornisce una serie di classi predefinite che permettono di gestire in modo piuttosto semplice l'interazione con l'utente e la visualizzazione dei dati utilizzando il sistema a finestre messo a disposizione dal sistema operativo utilizzato.

B.7.1 Struttura 'Model-View-Controller'

Tutta la comunicazione attraverso l'interfaccia grafica si basa sulla creazione e gestione dei componenti della struttura '*Model-View-Controller*' (MVC)

caratteristica di Smalltalk-80.

Model Implementa le funzionalità dell'applicazione che si vuole realizzare e quindi è in grado di fornire i dati da visualizzare o di ricevere le informazioni date dall'utente.

View Descrive come debbano essere visualizzate le informazioni o, più in generale, come appaia l'interfaccia tramite cui l'utente è in grado di comunicare con l'applicazione, cioè con il 'model'. Le 'view' possono essere inserite nelle finestre nel numero desiderato specificandone la posizione all'interno dell'area visibile.

Controller Gestisce la visualizzazione della 'view' e l'interazione con l'utente attraverso la tastiera o il 'mouse' facendo in modo che l'interfaccia reagisca alle azioni dell'utente ed il modello venga aggiornato riguardo a queste azioni. È compito del controllore mantenere l'aspetto esteriore dell'interfaccia coerente con il 'model' associato, anche quando quest'ultimo cambia.

Per realizzare l'interfaccia di un'applicazione è necessario scrivere le classi corrispondenti a questi tre componenti provvedendole dei metodi necessari affinchè le istanze possano comunicare. Però Objectworks(r)/Smalltalk fornisce tutta una serie di classi predefinite le quali implementano molti tipi di 'view' ed i loro corrispondenti 'controller' che possono essere utilizzate direttamente per realizzare svariate interfacce grafiche di utillità comune. Esistono anche 'model' predefiniti utilizzabili per elementi dell'interfaccia si tipo standard quali zone di inserimento teso, bottoni, liste, ma nella maggior parte dei casi il 'model' va scritto appositamente perché strettamente legato all'applicazione in via di sviluppo.

B.7.2 La realizzazione dei 'menu'

L'interfaccia standard di Objectworks(r)/Smalltalk prevede che alle 'view' possano essere associati dei 'menu' visualizzati quando l'utente preme il bottone centrale del 'mouse' mentre il puntatore si trova nell'area occupata dalla 'view' in questione. Questo 'menu' è gestito dal 'controller' associato, ma sia i comandi che contiene, che le azioni da intraprendere in corrispondenza della selezione dei vari comandi, possono essere stabiliti dal programmatore.

Infatti quando il 'controller' deve visualizzare il 'menu' lo chiede al 'model' associato mediante l'invocazione di un metodo il cui selettore è fornito al momento della creazione della 'view' in questione (vedi sezione successiva).

Un 'menu' è un'istanza della classe `PopUpMenu` che viene creata mediante l'invocazione di un metodo che ha come argomento due liste di simboli: quelli della prima sono i nomi dei comandi che appariranno nel 'menu', mentre quelli della seconda sono i selettori dei metodi da invocare sul 'model' associato alla 'view', quando il corrispondente comando viene selezionato.

B.7.3 La selezione in una lista

Una delle azioni più frequentemente richieste nell'esecuzione dell'esempio di processo, è la selezione di un elemento all'interno di una lista. Objectworks(r)/Smalltalk fornisce alcune classi predefinite che permettono di gestire in modo semplice questo tipo di interazione.

Si vuole far apparire su una finestra un riquadro contente una lista di cui è possibile far scorrere gli elementi e selezionarne uno. Questo è ottenibile creando un'istanza della classe `SelectionInListView` che è un tipico caso di componente 'view'. Il metodo di istanziazione può essere invocato con un messaggio del tipo riportato nella figura B.4; un esempio di invocazione si trova nella sezione 13.1.2 trattando della realizzazione di una delle interfacce utente di S^3.

,

```
agendaView := SelectionInListView
         noDelimitersOn:  aModel
         aspect:  aSymbol
         change:  changeMethodName:
         list:  listMethodName
         menu:  menuMethodName
         initialSelection:  selMethodName
```

Figura B.4. Selettore del metodo di istanziazione di `SelectionInList-View`

La prima 'keyword' indica semplicemente come deve apparire la lista dal momento che sono possibili alcune visualizzazioni alternative ottenibili invocando metodi differenti; l'argomento che segue ad essa è l'identificatore

del 'model' a cui fa riferimento. Il 'controller' è associato automaticamente ed è un'istanza della classe `SelectionInListController` predefinita da Objectworks(r)/Smalltalk.

Meccanismo di dipendenza

La 'view' ed il 'model' vengono collegati mediante il meccanismo di dipendenza implementato nella classe `Object` di Objectworks(r)/Smalltalk; questo fornisce un metodo la cui invocazione su di una istanza causa la notifica di tutti gli oggetti dipendenti di tale istanza. Questo metodo viene utilizzato prevalentemente per segnalare un cambiamento e quindi il messaggio comunemente utilizzato è

```
self changed:  #aspect
```

Cioè, quando un oggetto vuole notificare un cambiamento avvenuto al suo interno ai suoi dipendenti, invoca questo metodo su se stesso passando come argomento un simbolo che identifica quale dei suoi aspetti è cambiato. L'esecuzione di questo metodo causa l'invio di messaggi di notifica ai vari dipendenti fornendo `#aspect` come argomento. Infatti un'istanza può avere molti dipendenti e non tutti per gli stessi motivi, per cui non è detto che il cambiamento interessi tutti i dipendenti che decideranno se e come reagire al cambiamento in base al simbolo che ricevono.

Proprio questo meccanismo è utilizzato per comunicare alla 'view' che visualizza la lista un cambiamento nella lista stessa, che è dovuto ad un cambiamento nel modello associato. Poiché però il modello può cambiare sotto vari aspetti, è necessario comunicare alla 'view' qual è l'aspetto che la interessa e ciò è fatto mediante l'argomento `aSymbol` che segue la parola chiave `aspect:`. Ogni volta che il modello cambia in qualcosa che interessa la 'view' in questione invoca su sé stesso il metodo `changed:` passando come argomento proprio `aSymbol`. Dal canto suo la 'view' reagisce solo alle notifiche di cambiamento che riguardano l'aspetto `aSymbol`.

Comunicazione da 'view' a 'model'

Quando alla 'view' viene comunicato un cambiamento nel modello, essa deve poter ricavare la nuova lista da visualizzare e per questo le è fornito l'argomento `listMethodName` che è un simbolo coincidente con il selettore di un metodo del modello da invocare per avere, come valore di ritorno, la nuova

lista. Questo metodo viene anche invocato la prima volta che la lista deve essere visualizzata.

Se invece l'utente opera una selezione o una deselezione sulla lista, il modello associato ne deve essere informato in modo da gestire opportunamente l'evento (ad esempio aggiornando una variabile contenente l'elemento correntemente selezionato). Questo viene fatto invocando il metodo il cui selettore è individuato dal simbolo `changeMethodName` che segue la parola chiave `change:`.

Infine è possibile associare all'interfaccia un 'menu' che viene mostrato quando l'utente preme l'apposito pulsante del 'mouse' con il puntatore sulla zona dello schermo occupata dall'interfaccia in questione. Il 'controller' associato alla 'view' ottiene questo 'menu' come valore di ritorno dell'invocazione del metodo il cui selettore coincide con il simbolo `menuMethodName` fornito come argomento dopo la parola chiave `menu:`, e lo visualizza.

`selMethodName` è un simbolo che coincide con il selettore del metodo invocato dalla 'view' sul 'model' associato per ottenere il numero dell'elemento che deve essere selezionato quando la lista viene visualizzata la prima volta; se non deve essere selezionato nessun elemento, si usa `nil` come argomento.

B.7.4 La visualizzazione delle finestre

Le 'view' possono essere visualizzate inserendole all'interno di finestre precedentemente create. La classe `ScheduledWindow` fornisce il supporto per la gestione delle finestre le quali appaiono nel formato standard del sistema operativo utilizzato. Ad ogni istanza, creata con il metodo `new`, deve essere associata una 'view' mediante il metodo `component:` la quale viene visualizzato nella finestra corrispondente. Spesso può essere utile visualizzare più 'view' sulla stessa finestra (ad esempio un certo numero di liste) e ciò può essere fatto includendoli in un oggetto di tipo `CompositePart` che è in seguito associato alla finestra mediante `component:`.

Quando si crea un'istanza della classe `ScheduledWindow`, le viene automaticamente associato un controllore che gestisce gli eventi che la riguardano e che è un'istanza della classe `StandardSystemController`; inviando all'oggetto interesato il messaggio `controller:` è possibile assocare alla finestra un altro tipo di controllore.

Uno dei compiti fondamentali del controllore è la gestione del 'menu' che

è associato ad ogni finestra permettendo la chiusura di questa, il suo ridimensionamento ed altre funzionalità. Esso viene visualizzato quando l'utente preme il bottone a sinistra del 'mouse' con il puntatore nell'area di schermo occupata dalla finestra. Nella sezione 13.1.2 è mostrato un caso in cui al controllore standard è sostituito un altro controllore (`NoCloseController` al fine di avere un 'menu' associato alla finestra che non consenta di chiuderla.

Dopo aver creato un'istanza di `ScheduledWindow` ed averle assegnato le 'view' che deve contenere, è necessario visualizzarla medinate l'invio di un apposito messaggio che specifica le coordinate dello schermo in cui deve essere posizionato il vertice in alto a sinistra, e la dimensione del rettangolo che la contiene. Il tipo di metodo invocato per la visualizzazione stabilisce anche se l'esecuzione dei comandi succesivi all'invocazione debba continuare o se sia bloccata perché il flusso di esecuzione che ha inviato il messaggio viene utilizzato dal controllore per la gestione della finestra.

B.8 Concorrenza

Pur non offrendo supporto particolare per gli oggetti attivi, Objectworks(r)/Smalltalk prevede una serie di classi e di metodi da utilizzare per la gestione della concorrenza. È infatti possibile creare un nuovo processo ed eseguirlo, compatibilmente con la disponibilità del processore, parallelamente a quello che lo ha creato. Per effettuare questa operazione si deve utilizzare il metodo `fork` della classe `BlockClosure` che permette di eseguire in un nuovo processo concorrente il contenuto di un blocco. Quello descritto non è l'unico modo per creare nuovi flussi di esecuzione, ma è quello che è stato utilizzato nell'implementazione di S^3.

B.8.1 La classe `Semaphore`

Quando si ha a che fare con processi concorrenti, ci si trova ad affrontare due problemi classici:

1. la sincronizzazione tra i vari processi;
2. la condivisione sicura dei dati.

La letteratura propone varie soluzioni e Smalltalk ne adotta una di quelle basilari, cioè l'uso dei *semafori*. A questo scopo Objectworks(r)/Smalltalk

mette a disposizione del programmatore la classe `Semaphore` le cui istanze permettono di realizzare le funzioni tipiche dei semafori.

B.8.2 La sincronizzazione dei processi

Il principale scopo per cui si sono creati i semafori è la sincronizzazione dei processi. Infatti un processo si può mettere in attesa su un semaforo e bloccare così la sua esecuzione, ovviamente senza consumare tempo di processore, fino a che un altro processo non lo sveglia. Secondo la terminologia classica associata ai semafori, l'accodamento viene realizzato mediante l'invocazione di una procedura detta *wait*, mentre il risveglio di uno dei processi bloccati avviene mediante la procedura *signal*. Ovviamente queste procedure devono fare riferimento ad un ben determinato semaforo perché in un sistema ne possono essere utilizzati molti.

L'implementazione ad oggetti di questo meccanismo è immediata definendo per la classe `Semaphore` i metodi `wait` e `signal` che agiscono esattamente nel modo descritto sopra.

B.8.3 La mutua escusione

Esiste una tecnica, universalmente nota e consolidata, che consente di utilizzare i semafori ed i loro due metodi fondamentali, per la protezione dei dati. Questo problema consta nel fatto che due processi non possono utilizzare lo stesso dato nello stesso momento, perché si riscierebbe che ciò porti a situazioni di incoerenza. Allora è necessario che, mentre un processo utilizza un dato condiviso da più processi, tutti gli altri restino bloccati finché quello in questione non ha terminato.

Un approccio molto semplice per la soluzione di questo problema consta nel creare una *regione critica* nel codice di un programma alla quale è associato un semaforo. L'attraversamento di questa regione impone il rispetto di un protocollo che consiste nella chiamata della funzione *wait* del semaforo associato, al momento dell'ingresso, e della funzione *signal*, al momento dell'uscita. Inizializzando il semaforo inivocando la funzione *signal* prima di cominciare l'esecuzione, si garantisce la *mutua esclusione*, cioè che non ci sia mai più di un processo all'interno delle regioni critiche associate ad un certo semaforo. Infatti, il primo processo che si trova ad entrare, esegue una *wait* e

non resta bloccato grazie alla *signal* invocata al momento dell'inizializzazione. Se un altro processo tenta di eseguire una regione critica associata allo stesso semaforo, resterà bloccato fino a che il processo che la sta eseguendo, uscendo, chiama la *signal* riattivando il primo dei processi in attesa sul semaforo coinvolto. Se alcuni dati sono manipolati solo all'interno di regioni critiche associate allo stesso semaforo, non c'è il rischio di avere situazioni pericolose di accesso.

B.8.4 Le regioni critiche

Objectworks(r)/Smalltalk fornisce un efficiente supporto all'utilizzo delle regioni critiche che si basa sul metodo `critical:` invocabile sugli oggetti della classe `Semaphore`. Quando si vuole utilizzare un semaforo per gestire regioni critiche si deve creare un'istanza della classe `Semaphore` invocando su quest'ultima il metodo `forMutualExclusion`, invece che quello standard `new`; infatti in questo modo l'oggetto viene inizializzato opportunamente. Le regioni critiche si possono definire racchiudendo il codice che le costituisce in un blocco, cioè tra parentesi quadre. Quindi si invoca il metodo `critical:` dell'oggetto `Semaphore` associato alla regione, fornendo il blocco in questione come argomento.

L'implementazione della regione critica da parte della classe `Semaphore` è esattamente quella descritta nella sezione precedente:

- quando si crea un'istanza della classe `Semaphore` mediante il metodo `forMutualExclusion` viene inviato un messaggio `signal` al nuovo oggetto;
- quando si invoca il metodo `critical:`, prima di eseguire il blocco passato come argomento, è mandato il messaggio `wait` e dopo l'esecuzione (o all'uscita dal blocco per effetto di un comando di ritorno da metodo) è mandato il messaggio `signal`.

In alcune situazioni piuttosto particolari è necessario liberare la regione critica prima di inviare un messaggio ad un altro oggetto per evitare di creare attese cicliche (vedi sezione 12.3.3). Bisogna tener presente che questa tecnica deve essere usata solamente per il trattamento di casi molto particolari, in quanto è in contrasto con il tipo di utilizzo per cui è stata pensata la regione critica.

Per come è implementato il meccanismo delle regioni critiche in Objectworks(r)/Smalltalk, questo effetto è ottenuto invocando il metodo `signal` del semaforo associato; però ciò va fatto con estrema cautela e soprattutto, prima della fine della regione, deve assolutamente avvenire l'invio di un messaggio `wait` che sancisce il fatto che la regione è nuovamente stata occupata. Infatti se ciò non accade al momento dell'uscita dalla regione critica si creano problemi che possono portare ad avere in seguito due processi che entrano in regioni protette dal semaforo in questione.

Appendice C

Definizione delle classi di S^3

C.1 La categoria Task

```
Controlled subclass: #Task
   instanceVariableNames: 'state toBeRestarted
associatedTools father children assignedRoles responsible
successors predecessors waitingFeedback input output
userAction '
   classVariableNames: 'Children Input Output Predecessors
ResponsibleRole WaitingFeedback '
   poolDictionaries: ''
   category: 'Task'



Task methodsFor: initialization

initialize: parent
   "initialize all instance variables"

   children := BooleanInstanceRelation new.
   predecessors := BooleanInstanceRelation new.
   successors := InstanceConnection new.
   waitingFeedback := InstanceConnection new.
   input := InstanceConnection new.
```

```
   output := InstanceConnection new.
   father := parent.
   toBeRestarted := false.
   userAction := Semaphore new.

Task methodsFor: standard

again
   "reactivate the task"

   self critical: [state = #executing
         ifTrue: [toBeRestarted := true]
         ifFalse: [self changeState: #waitingPreconditions]]

changeState: newState
   "check if it is possible to change status"

   newState = #waitingPreconditions & (state = #completed) |
(newState = #completed)
      ifTrue:
         [state := newState.
         successors do: [:taskId | [taskId handleTransition:
self] fork].
         [father handleTransition: self] fork]
      ifFalse:
         [newState = #executing
            ifTrue:
               [state := newState.
               self startExecution.
               ^self].
         toBeRestarted
            ifTrue:
               [toBeRestarted := false.
               state := #waitingPreconditions]
            ifFalse: [newState notNil ifTrue: [state :=
newState]]].
   #waitingPreconditions = state
```

```
      ifTrue:
         [predecessors allCompleted ifFalse: [^self].
         self changeState: #executing.
         ^self].
   #handlingChildren = state ifTrue: [predecessors
allCompleted
         ifTrue:
            [Children childrenAndNumberOf: self class do:
[:childClass :num | num > 0 ifTrue: [(children
instancesNumberOfClass: childClass) isZero ifTrue:
[^self]]].
            children allCompleted
               ifTrue:
                  [self changeState: #completed.
                  ^self]]
         ifFalse: [self changeState:
#waitingPreconditions]].
   #completed = state & (predecessors allCompleted &
children allCompleted) not ifTrue: [self changeState:
#waitingPreconditions]

fromUser: matter
   "the Responsible is available for matter"

   matter = #terminate ifTrue: [self terminateExecution].
   matter = #toolChoice | (matter = #redo) ifTrue:
[userAction signal].
   #fail = matter ifTrue: [self giveFeedback]

giveFeedback
   "send feedback to waiters and children"

   | tmp |
   tmp := Semaphore new.
   waitingFeedback do: [:waiter |
      [waiter again.
      tmp signal] fork].
```

```
   self signal.
   tmp wait.
   self wait.
   self changeState: #waitingPreconditions

gotUserAttention
   "the Responsible is available selecting the activity from
Agenda"

   userAction signal

handleTransition: sender
   "declares the task completed"

   self
      critical:
         [(children includesKey: sender)
            ifTrue: [children at: sender put: (children at:
sender) not]
            ifFalse: [predecessors at: sender put:
(predecessors at: sender) not].
         self instantiateChildren.
         self changeState: nil]

startExecution
   "executes task"

   self createTemporalOutput.
   self reConnectInput.
   self restartChildren.
   self instantiateChildren

terminateExecution
   "declares the execution completed"

   self removeTemporalOutput.
   self changeState: #handlingChildren
```

```
Task methodsFor: relation handling

addFeedbackWaiter: aTask
   "add a task WaitingFeedback"

   self critical: [waitingFeedback add: aTask]

addPredecessor: aTask
   "add a Predecessor"

   [aTask addSuccessor: self] fork.
   self critical: [predecessors addTrue: aTask]

addSuccessor: aTask
   "add a Successor"

   self critical: [successors add: aTask]

associatedTools: toolInstanceRelation
   "connects the new task to its associated tools"

   self critical: [associatedTools := toolInstanceRelation
copy]

connectRoles: rolesInstanceRelation
   "connects rolesInstanceRelation roles"

   self critical: [assignedRoles := rolesInstanceRelation]

endOfAssignation
   "states that all assignations are done"

   self critical: [self changeState: #waitingPreconditions]

inputIs: anInstanceConnection
   "connect the Task to its input"
```

```
   self critical: [input := anInstanceConnection]

iOConditioningData
   "return data conditioning I/O"

   | tmp |
   tmp := Set new.
   (Array with: input with: output)
      do: [:sender | sender do:
            [:data |
            tmp addAll: (data conditioningDataFrom: data).
            tmp add: data]].
   ^tmp

iORelatedData
   "return data related with I/O"

   | tmp |
   tmp := Set new.
   (Array with: input with: output)
      do: [:sender | sender do:
            [:data |
            tmp addAll: (data relatedDataFrom: self).
            (TemporarySwObject subclasses includes: data
class)
               ifFalse: [tmp add: data]]].
   ^tmp

output
   "return a set containing all software objects in input
and in output"

   | tmp |
   self critical: [tmp := output copy].
   ^tmp
```

```
outputIs: anInstanceConnection
   "connect the Task to its output"

   self critical: [output := anInstanceConnection].
   anInstanceConnection do: [:data | data productorIs: self]

outputName
   output do: [:aData | (TemporarySwObject subclasses
includes: aData class)
         ifFalse: [^aData provideName]]

reConnectInput
   "connect IO for newChild.
   The method connects instances of classes specified by
Input and Output
   newChild's class relations that are connected by input or
output Instance
   Connection of self"

   | related |
   related := self iORelatedData.
   input := InstanceConnection new.
   self class input do: [:dataClass | input addAll: (related
instancesOfClass: dataClass)]

responsibleIs: resp
   "connects a task to its Responsible"

   self critical: [responsible := resp]

Task methodsFor: private

askAttentionFor: aString1 description: aString2
   "ask responsible attention"

   responsible
      recordActivity: aString1
```

```
      from: self
      description: aString2.
   self signal.
   userAction wait.
   self wait

chooseTools
   "if necessary requires to the user for the tool he
prefers to perform the task"

   | collection inserted |
   inserted := Set new.
   collection := TripleArray new.
   output do: [:data | data provideTool isNil ifTrue:
         [| toolClass |
         (toolClass := data class provideTool) notNil
ifTrue:
            [| chosen tmp |
            ((tmp := associatedTools instancesOfClass:
toolClass) includes: (chosen := responsible choiceOfClass:
toolClass))
               ifTrue: [data creatorIs: chosen]
               ifFalse: [tmp size > 1 ifFalse: [tmp size
isZero
                        ifTrue: [WarningWindow text: 'Any
tool of class ' , toolClass provideName , ' present.
Probable instantiation error!!']
                        ifFalse: [tmp do: [:tool | data
creatorIs: tool]]]
                     ifTrue: [(inserted includes: toolClass)
                           ifFalse:
                              [| temp |
                              inserted add: toolClass.
                              temp := DoubleArray new.
                              tmp do: [:tool | temp add:
tool provideName with: tool].
                              collection
```

```
                                    add: toolClass provideName
                                    with: temp
                                    and: toolClass]]]]]].
   collection size isZero
      ifFalse:
         [DoubleListSelection
            on: collection
            for: self
            windowTitle: 'Tool Choice'
            firstListLabel: 'Tool Category'
            secondListLabel: 'AvailableTools'.
         self signal.
         userAction wait.
         self wait.
         inserted do: [:test | ^(responsible choiceOfClass:
test) notNil]].
   ^true

createTemporalOutput
   "create output that has to be alive until task correct
termination.
   This is data whose class is connected by Output class
variable, and is
   subclass of TemporalOutput"

   self class output do: [:dataClass | (TemporarySwObject
subclasses includes: dataClass)
         & (output instancesOfClass: dataClass) isEmpty
         ifTrue:
            [| tmp temp |
            tmp := dataClass new is: 'by_' , self getName.
            output add: tmp.
            temp := Set new addAll: input.
            output do: [:aData | temp addAll: (aData
conditioningDataFrom: self)].
            temp do: [:aData | tmp relatedDataIs: aData]]
```

```
getName
   "return name for private invocation"

   ^self class printString

openTerminationInterface
   "opens an interface by which the user can specify
termination type"

   FailableTermination of: self getName inTask: self

removeTemporalOutput
   "remove output that has to be alive until task correct
termination.
   This is data whose class is connected by Output class
variable, and is
   subclass of TemporalOutput"

   (output instancesOfSubclassOf: TemporarySwObject)
      do:
         [:data |
         output remove: data.
         data remove]

restartChildren
   | tmp flag |
   tmp := Semaphore new.
   flag := false.
   Children everyChildrenOf: self class do: [:childClass |
childClass predecessorsNumber isZero ifTrue: [(children
instancesOfClass: childClass)
            do:
               [:childId |
               flag := true.

               [childId again.
               tmp signal] fork]]].
```

```
   flag
      ifTrue:
         [self signal.
         tmp wait.
         self wait]

runTools
   "run appropriate tools on input and output data"

   [self chooseTools]
      whileFalse: [self askAttentionFor: 'Choose tools for '
, self getName description: 'Make tool choice and go on
performing activity'].
   (Array with: input with: output)
      do: [:receiver | receiver do:
            [:data |
            | tool |
            (tool := data provideTool) notNil ifTrue: [tool
runOn: data]]]

Task methodsFor: communication

fail
   self critical: [self giveFeedback]

isThereResponsible
   "return true if the task has a responsible"

   | tmp |
   self critical: [tmp := responsible notNil].
   ^tmp

provideAssignedRoles
   "return assignedRoles"

   | tmp |
   self critical: [tmp := assignedRoles].
```

```
   ^tmp

provideName
   "return a name identifying the task"

   | tmp |
   self critical: [tmp := self getName].
   ^tmp

selectionIs: selection
   "the DoubleListSelection interface returns the selection"

   self critical: [selection notNil
         ifTrue:
            [selection values do: [:tool | responsible
choiceIs: tool].
            output do: [:data | data provideTool isNil &
data class provideTool notNil ifTrue: [data creatorIs:
(selection at: data class provideTool)]]]].
   userAction signal

suspend
   self critical: [self startExecution]

terminate
   self critical: [self terminateExecution]

Task methodsFor: children instantiation

assignFor: childId
   "make assignation for a child if possible, else it defers
everything to a
   AssignTasks task"

   | neededRoleClasses rolesToConnect playingPersons
needAssignTasks |
   needAssignTasks := false.
```

```
   rolesToConnect := InstanceConnection new.
   neededRoleClasses := childId class childrenNeededRoles.
   neededRoleClasses do:
      [:roleClass |
      playingPersons := (assignedRoles instanceOfClass:
roleClass) players.
      playingPersons size == 1
         ifTrue: [rolesToConnect add: (roleClass new:
playingPersons)]
         ifFalse:
            [needAssignTasks := true.
            rolesToConnect add: (roleClass new:
InstanceConnection new)]].
   childId class responsibleRole isNil
      ifTrue: [childId responsibleIs: responsible]
      ifFalse:
         [playingPersons := (assignedRoles instanceOfClass:
childId class responsibleRole) players.
         playingPersons size == 1
            ifTrue: [playingPersons do: [:pers | childId
responsibleIs: pers]]
            ifFalse: [needAssignTasks := true]].
   childId connectRoles: rolesToConnect.
   needAssignTasks ifFalse: [[childId endOfAssignation]
fork]
      ifTrue:
         [| assigner |
         (assigner := children instanceOfClass: AssignTasks)
isNil
            ifFalse:
               [assigner assign: childId.
               [assigner again] fork]
            ifTrue: [children add: ((AssignTasks
                     new: self
                     responsible: responsible
                     assign: childId)
                     connectRoles: assignedRoles)]]
```

```
connectIOOf: newChild
   "connect IO for newChild.
   The method connects instances of classes specified by
Input and Output
   newChild's class relations that are connected by input or
output Instance
   Connection of self"

   | data |
   data := InstanceConnection new.
   newChild class input do: [:dataClass | data addAll: (self
iORelatedData instancesOfClass: dataClass)].
   newChild inputIs: data.
   data := InstanceConnection new.
   newChild class output do: [:dataClass | data addAll:
(self iORelatedData instancesOfClass: dataClass)].
   newChild outputIs: data

connectSiblingsOf: newChild
   "connect predecessors and feedback waiters"

   | conditioningTasks |
   conditioningTasks := Set new.
   newChild output do: [:data | conditioningTasks addAll:
data conditioningTasks].
   conditioningTasks := self onlyChildrenAmong:
conditioningTasks.
   newChild class predecessorsDo:
      [:predClass |
      | tmp |
      (tmp := conditioningTasks instancesOfClass: predClass)
size isZero
         ifTrue: [(children instancesOfClass: predClass)
               do: [:predId | newChild addPredecessor:
predId]]
         ifFalse: [tmp do: [:task | newChild addPredecessor:
```

```
task]]].
   newChild class feedbackWaitersDo:
      [:waiterClass |
      | tmp |
      (tmp := conditioningTasks instancesOfClass:
waiterClass) size isZero ifFalse: [tmp do: [:aTask |
newChild addFeedbackWaiter: aTask]]
         ifTrue: [(children instancesOfClass: waiterClass)
               do: [:waiterId | newChild addFeedbackWaiter:
waiterId]]]

instantiate: number childrenOfClass: childrenClass
   "instantiates number children of the class childrenClass"

   number
      timesRepeat:
         [| newChild |
         children add: (newChild := childrenClass new:
self).
         self connectIOOf: newChild.
         newChild associatedTools: associatedTools.
         self connectSiblingsOf: newChild.
         self assignFor: newChild]

instantiateChildren
   "instantiates children if possible"

   Children childrenAndNumberOf: self class do: [:childClass
:num | (num isZero
         ifTrue: [self
isPossibleToInstantiateChildrenOfClass: childClass]
         ifFalse: [self
isPossibleToInstantiateFixedChildrenOfClass: childClass])
         ifTrue: [num isZero
               ifTrue: [self instantiateChildrenOfClass:
childClass]
               ifFalse: [self instantiate: num
```

```
childrenOfClass: childClass]]]

instantiateChildrenOfClass: childrenClass
   "instantiates children of the class childrenClass.
   This method is done for tasks having at most one output
object"

   childrenClass output do: [:dataClass | (self
iOConditioningData instancesOfSubclassOf: dataClass)
         do: [:dataObj | dataObj productors size isZero
ifTrue: [(self isPossibleToInstantiateChildOfClass:
childrenClass withConditioningData: (dataObj
conditioningDataFrom: self))
                  ifTrue:
                     [| newChild dataIn conditioningData |
                     children add: (newChild :=
childrenClass new: self).
                     newChild outputIs: (InstanceConnection
new add: dataObj).
                     dataIn := InstanceConnection new.
                     conditioningData := dataObj
conditioningDataFrom: dataObj.
                     childrenClass input do:
                        [:dataInClass |
                        | tmp |
                        (tmp := conditioningData
instancesOfClass: dataInClass) isEmpty ifFalse: [dataIn
addAll: tmp]].
                     newChild inputIs: dataIn.
                     newChild associatedTools:
associatedTools.
                     self connectSiblingsOf: newChild.
                     self assignFor: newChild]]]].
   ^self

isPossibleToInstantiateChildOfClass: childClass
withConditioningData: data
```

```
   "check if all predecessors are completed"

   childClass
      predecessorsDo:
         [:predClass |
         | tmp |
         tmp := Set new.
         data do:
            [:each |
            | temp |
            (temp := each predecessorTasksOfClass:
predClass) isNil ifTrue: [^false].
            tmp addAll: (self onlyChildrenAmong: temp)].
         (tmp isEmpty
            ifTrue: [(children allCompletedOfClass:
predClass)
                  & (children instancesOfClass: predClass)
isEmpty not]
            ifFalse: [children allCompleted: tmp])
            ifFalse: [^false]].
   ^true

isPossibleToInstantiateChildrenOfClass: childClass
   "test if is possible to instantiate children of class
childClass"

   | inputConditioningData |
   inputConditioningData := Set new.
   childClass input do: [:dataClass | (self
iOConditioningData instancesOfSubclassOf: dataClass)
         do: [:data | inputConditioningData addAll: (data
conditioningDataFrom: self)]].
   childClass output do: [:dataClass | (self
iOConditioningData instancesOfSubclassOf: dataClass)
         do: [:dataObj | dataObj productors isEmpty
               ifTrue:
                  [| conditioningData |
```

```
                    conditioningData := inputConditioningData
copy.
                    conditioningData addAll: (dataObj
conditioningDataFrom: self).
                    (self isPossibleToInstantiateChildOfClass:
childClass withConditioningData: conditioningData)
                        ifTrue: [^true]]]].
    ^false

isPossibleToInstantiateFixedChildrenOfClass: childClass
    "test if is possible to instantiate children of class
childClass"

    | conditioningData |
    (children instancesOfClass: childClass) isEmpty not
ifTrue: [^false].
    conditioningData := Set new.
    childClass inputOutput do: [:dataClass | (self
iOConditioningData instancesOfClass: dataClass)
          do: [:data | conditioningData addAll: (data
conditioningDataFrom: self)]].
    ^self isPossibleToInstantiateChildOfClass: childClass
withConditioningData: conditioningData

onlyChildrenAmong: taskSet
    "return a set containing only elements of taskSet that
are children of self"

    | tmp |
    tmp := Set new.
    taskSet do: [:task | (children includes: task)
          ifTrue: [tmp add: task]].
    ^tmp


Task class
    instanceVariableNames: ''
```

```
Task class methodsFor: instance creation

new: father
   "creates a new instance of the task and initializes it"

   | newInst |
   newInst := super new.
   newInst initialize: father.
   ^newInst

Task class methodsFor: relation handling

childrenNeededRoles
   "provide his children's ResponsibleRoles"

   | roles |
   roles := Set new.
   Children everyChildrenOf: self do: [:childClass |
childClass neededRoles do: [:neededRole | roles add:
neededRole]].
   ^roles

feedbackWaitersDo: block
   "executes block on each occurence of WaitingFeedback"

   ^WaitingFeedback of: self do: block

input
   "return Input class relation"

   ^(Input of: self) copy

inputOutput
   "return Input class relation concatenated with Output
```

```
class relation"

   | tmp |
   tmp := Set new.
   tmp addAll: (Input of: self).
   tmp addAll: (Output of: self).
   ^tmp

neededRoles
   "provide his ResponsibleRole and his children's
ResponsibleRoles"

   | roles |
   roles := Set new.
   roles add: self responsibleRole.
   Children everyChildrenOf: self do: [:childClass |
childClass ~= self ifTrue: [childClass neededRoles do:
[:neededRole | roles add: neededRole]]].
   ^roles

output
   "return Input class relation concatenated with Output
class relation"

   ^(Output of: self) copy

predecessorsDo: block
   "executes block on each occurence of Predecessors"

   ^Predecessors of: self do: block

predecessorsNumber
   "returns the number of class of predecessors connected"

   ^Predecessors numberFor: self

responsibleRole
```

```
   "returns the class of the role to be played by the
responsible"

   ^ResponsibleRole of: self

Task class methodsFor: initialization

initialize
   "initializes class variables"

   Children := ChildrenClassRelation new.
   Predecessors := ClassRelation new.
   WaitingFeedback := ClassRelation new.
   ResponsibleRole := SingleClassRelation new.
   Input := ClassRelation new.
   Output := ClassRelation new.
   self allSubclasses do: [:subClass | subClass initialize]
```

```
Task subclass: #ReviewDesign
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Task'



ReviewDesign methodsFor: standard

startExecution
   "ask for user attention"

   super startExecution.
   self askAttentionFor: 'Review product design'
description: 'Review design of product in development'.
   self runTools.
   self openTerminationInterface

ReviewDesign methodsFor: private

runTools
   "run tools on input and output data"

   super runTools.
   WarningWindow text: 'Configuration editor opened on ' ,
(input instanceOfClass: Configuration) provideName


ReviewDesign class
   instanceVariableNames: ''



ReviewDesign class methodsFor: initialization
```

```
initialize
   Children new: self.
   Children
      of: self
      is: AssignTasks
      numberOfInstances: 0.
   Predecessors new: self.
   Predecessors of: self is: SelectTool.
   Predecessors of: self is: Design.
   WaitingFeedback new: self.
   WaitingFeedback of: self is: Design.
   ResponsibleRole of: self is: DesignReviewer.
   Input new: self; of: self is: Configuration; of: self is:
RequirementDocument; of: self is: DesignDocument.
   Output new: self; of: self is: FeedbackDocument
```

```
Task subclass: #Design
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Task'



Design methodsFor: communication

provideName
   "return a neme identifying the task"

   ^'Design'

Design methodsFor: standard

startExecution
   "ask for user attention"

   | config |
   super startExecution.
   ((config := output instanceOfClass: Configuration)
includes: DesignDocument)
      ifFalse:
         [| temp |
         temp := DesignDocument new is: 'Design document'.
         (config componentsOfClass: RequirementDocument)
            do: [:req | temp meet: req].
         config include: temp.
         output add: temp.

         ["automatic configuration construction"
         | i1 s1 h s2 i2 i3 o1 o2 e c |
         c := CommentFile new is: 'Comment'.
         s1 := CompilableSourceFile new is: 'Source 1'.
```

```
          i1 := SourceFile new is: 'Include 1'.
          h := SourceFile new is: 'Header 1'.
          s2 := CompilableSourceFile new is: 'Source 2'.
          i2 := SourceFile new is: 'Include 2'.
          i3 := SourceFile new is: 'Include 3'.
          o1 := ObjectFile new is: 'Object 1'.
          o2 := ObjectFile new is: 'Object 2'.
          e := ExecutableFile new is: 'Program file'.
          c comment: s1.
          s1 use: h; use: i1.
          s2 use: h; use: i2; use: i3.
          s1 generate: o1.
          s2 generate: o2.
          o1 compose: e.
          o2 compose: e.   "end of configuration"
          config include: c; include: s1; include: i1;
include: h; include: s2; include: i2; include: i3; include:
o1; include: o2; include: e] value].
   self askAttentionFor: 'Design product' description: 'Make
design of product in development'.
   self runTools.
   self openTerminationInterface

Design methodsFor: private

openTerminationInterface
   "opens an interface by which the user can specify
termination type"

   NotFailableTermination of: 'Design product' inTask: self

runTools
   "run tools on input and output data"

   super runTools.
   WarningWindow text: 'Configuration editor opened on ' ,
(output instanceOfClass: Configuration) provideName
```

```
Design class
   instanceVariableNames: ''



Design class methodsFor: initialization

initialize
   Children new: self.
   Children
      of: self
      is: AssignTasks
      numberOfInstances: 0.
   Children
      of: self
      is: Design
      numberOfInstances: 0.
   Predecessors new: self.
   Predecessors of: self is: SelectTool.
   ResponsibleRole of: self is: Designer.
   Output new: self.
   Output of: self is: (Configuration addProductor: self).
   Input new: self; of: self is: RequirementDocument; of:
self is: FeedbackDocument
```

```
Task subclass: #Edit
   instanceVariableNames: 'outputStatus '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Task'



Edit methodsFor: private

openTerminationInterface
   "opens an interface by which the user can specify
termination type"

   NotFailableTermination of: 'Edit ' , self outputName
inTask: self

Edit methodsFor: standard

again
   "reactivate the task"

   self
      critical:
         [self restartChildren.
         state = #executing
            ifTrue: [toBeRestarted := outputStatus =
#modifying]
            ifFalse: [self changeState:
#waitingPreconditions]]

startExecution
   "ask for user attention"

   super startExecution.
   outputStatus := #notYetModified.
```

```
   self askAttentionFor: self getName description: 'Edit
file ' , self outputName.
   self runTools.
   outputStatus := #modifying.
   self openTerminationInterface

Edit methodsFor: relation handling

inputIs: anInstanceConnection
   "connect the Task to its input"

   self
      critical:
         [input isNil ifTrue: [input := InstanceConnection
new].
         input addAll: anInstanceConnection]

outputIs: anInstanceConnection
   "connect the Task to its output"

   super outputIs: anInstanceConnection.
   self critical: [anInstanceConnection do: [:data | input
addAll: (data relatedDataFrom: self)]]

Edit methodsFor: communication

getName
   "return a neme identifying the task"

   ^'Edit ' , self outputName


Edit class
   instanceVariableNames: ''
```

```
Edit class methodsFor: initialization

initialize
   ResponsibleRole of: self is: Programmer.
   Input new: self; of: self is: DesignDocument; of: self
is: FeedbackDocument.
   Output new: self; of: self is: (SourceFile addProductor:
self); of: self is: (CompilableSourceFile addProductor:
self); of: self is: (CommentFile addProductor: self)
```

```
Task subclass: #Program
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Task'



Program methodsFor: standard

startExecution
   "executes task"

   super startExecution.
   self terminateExecution


Program class
   instanceVariableNames: ''



Program class methodsFor: initialization

initialize
   Children new: self.
   Children
      of: self
      is: AssignTasks
      numberOfInstances: 0.
   Children
      of: self
      is: Edit
      numberOfInstances: 0.
   Children
      of: self
```

```
   is: Compile
   numberOfInstances: 0.
Children
   of: self
   is: LinkModules
   numberOfInstances: 0.
Predecessors new: self.
Predecessors of: self is: SelectTool.
Predecessors of: self is: Design.
ResponsibleRole of: self is: Designer.
Input new: self; of: self is: Configuration
```

```
Task subclass: #Compile
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Task'



Compile methodsFor: standard

startExecution
   "ask for user attention"

   super startExecution.
   self runTools: #noUser.
   (DialogView confirm: 'Do you want to simulate Compile
correct termination?')
      ifTrue: [self terminateExecution]
      ifFalse: [self editErrorReport]

Compile methodsFor: communication

getName
   "return a neme identifying the task"

   ^'Compile ' , self outputName

suspend
   self critical: [self editErrorReport]

terminate
   self critical: [self giveFeedback]

Compile methodsFor: private

chooseTools: aSymbol
```

```
   "if necessary requires to the user for the tool he
prefers to perform the task"

   | collection inserted |
   inserted := Set new.
   collection := TripleArray new.
   output do: [:data | data provideTool isNil ifTrue:
         [| toolClass |
         (aSymbol = #noUser | (aSymbol = #auto)
            ifTrue: [CompileTool withAllSubclasses includes:
(toolClass := data class provideTool)]
            ifFalse: [(toolClass := data class provideTool)
notNil])
            ifTrue:
               [| chosen tmp |
               ((tmp := associatedTools instancesOfClass:
toolClass) includes: (chosen := responsible choiceOfClass:
toolClass))
                  ifTrue: [data creatorIs: chosen]
                  ifFalse: [tmp size > 1 ifFalse: [tmp size
isZero
                           ifTrue: [WarningWindow text: 'Any
tool of class ' , toolClass provideName , ' present.
Probable instantiation error!!']
                           ifFalse: [tmp do: [:tool | data
creatorIs: tool]]]
                        ifTrue: [(inserted includes:
toolClass)
                              ifFalse:
                                 [| temp |
                                 inserted add: toolClass.
                                 temp := DoubleArray new.
                                 tmp do: [:tool | temp add:
tool provideName with: tool].
                                 collection
                                    add: toolClass
provideName
```

```
                                    with: temp
                                    and: toolClass]]]]]].
   collection size isZero
      ifFalse:
         [aSymbol = #noUser ifTrue: [^false].
         DoubleListSelection
            on: collection
            for: self
            windowTitle: 'Tool Choice'
            firstListLabel: 'Tool Category'
            secondListLabel: 'AvailableTools'.
         self signal.
         userAction wait.
         self wait.
         inserted do: [:test | ^(responsible choiceOfClass:
test) notNil]].
   ^true

editErrorReport
   "ask for user attention and run editor"

   self askAttentionFor: 'Report errors for task ' , self
getName description: 'Edit error report of file ' , self
outputName.
   self runTools: #userInteracting.
   self openTerminationInterface

openTerminationInterface
   "opens an interface by which the user can specify
termination type"

   NotFailableTermination of: 'Error report of ' , self
outputName inTask: self

runTools: aSymbol
   "run appropriate tools on input and output data"
```

```
   | parameter |
   parameter := aSymbol.
   [self chooseTools: parameter]
      whileFalse:
         [self askAttentionFor: 'Choose tools for ' , self
getName description: 'Make tool choice and go on performing
activity'.
         aSymbol = #noUser ifTrue: [parameter := #auto]].
   (Array with: input with: output)
      do: [:receiver | receiver do:
            [:data |
            | tool |
            (tool := data provideTool) notNil ifTrue:
[aSymbol ~= #userInteracting & (CompileTool
withAllSubclasses includes: tool class) | (aSymbol =
#userInteracting & (CompileTool withAllSubclasses includes:
tool class) not) ifTrue: [tool runOn: data]]]]


Compile class
   instanceVariableNames: ''



Compile class methodsFor: initialization

initialize
   Predecessors new: self.
   Predecessors of: self is: Edit.
   WaitingFeedback new: self.
   WaitingFeedback of: self is: Edit.
   ResponsibleRole of: self is: Programmer.
   Output new: self; of: self is: (ObjectFile addProductor:
self); of: self is: FeedbackDocument
```

```
Task subclass: #AssignTasks
   instanceVariableNames: 'toAssign '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Task'



AssignTasks methodsFor: relation handling

assign: taskId
   "connects the task taskId by its attribute toAssign"

   self critical: [toAssign add: taskId]

selectionIs: selection
   "the MultipleListMultipleSelection interface returns the
selection"

   self critical: [selection notNil
         ifTrue:
            [selection do:
               [:task :resp :roles |
               resp notNil ifTrue: [task responsibleIs:
resp].
               roles do:
                  [:role :personArray |
                  | players |
                  players := InstanceConnection new.
                  personArray do: [:person | players add:
person].
                  role players: players].
               toAssign remove: task.
               [task endOfAssignation] fork].
            self terminateExecution]
         ifFalse: [self startExecution]]
```

```
AssignTasks methodsFor: private

makeAssignableCollection
   "return the variable collection which has the following
structure:
   collection is a TripleArray. Each element is:
   a task name;       (A);                  (B)
   (A) is a TripleArray. Each element is:
   a role name;      (C) ;                  a role
identifier
   (B) is a 3 slots Array. Each slot contains:
   a task identifier;   a responsible role name   (D)
   (C) is a DoubleArray. Each element is:
   a person name   a person identifier {playing related
role}
   (D) is a DoubleArray. Each element id:
   a person name   a person identifier {playing responsible
role}
   "

   | collection |
   collection := TripleArray new.
   toAssign do:
      [:taskId |
      | roles responsibleRoleName personList |
      taskId provideAssignedRoles do:
         [:role |
         roles := TripleArray new.
         role players size isZero
            ifTrue:
               [| persons |
               persons := DoubleArray new.
               (assignedRoles instanceOfClass: role class)
players do: [:person | persons add: person provideName with:
person].
               role class = self class responsibleRole
```

```
ifTrue: [persons add: responsible provideName with:
responsible].
               roles
                  add: role class provideName
                  with: persons
                  and: role]].
      personList := DoubleArray new.
      taskId isThereResponsible
         ifFalse:
            [(assignedRoles instanceOfClass: taskId class
responsibleRole) players do: [:person | personList add:
person provideName with: person].
            responsibleRoleName := taskId class
responsibleRole provideName]
         ifTrue: [responsibleRoleName := nil].
      collection
         add: taskId provideName
         with: roles
         and: (Array
               with: taskId
               with: responsibleRoleName
               with: personList)].
   ^collection

openSelectionInterface
   "opens selection interface"

   | assignableColl |
   assignableColl := self makeAssignableCollection copy.
   MultipleListMultipleSelection
      on: assignableColl
      for: self
      windowTitle: 'Task Assignation'
      labels: ((Array
            with: 'Task to assign'
            with: 'roles played'
            with: 'responsible role'
```

```
            with: 'Available persons')
            add: 'Choosen persons';
            add: 'Available persons')

AssignTasks methodsFor: standard

startExecution
   "start execution of task body"

   toAssign isEmpty
      ifTrue: [self terminateExecution]
      ifFalse:
         [super startExecution.
         self askAttentionFor: 'Assign Tasks' description:
'Select responsibles for shown tasks'.
         self openSelectionInterface]

terminateExecution
   "if possible terminate execution"

   toAssign isEmpty ifTrue: [toBeRestarted := false].
   super terminateExecution

AssignTasks methodsFor: initialization

initialize: parent responsible: resp assign: taskId
   "initializes new instance by connecting it to its
responsible and its father and
   initializing other instance variables"

   super initialize: parent.
   responsible := resp.
   (toAssign := Set new) add: taskId.
   [self endOfAssignation] fork.
   ^self
```

```
AssignTasks class
   instanceVariableNames: ''



AssignTasks class methodsFor: initialization

initialize
   "does not do anything"

AssignTasks class methodsFor: instance creation

new: father responsible: resp assign: taskId
   "creates a new instance of the task"

   ^super new
      initialize: father
      responsible: resp
      assign: taskId
```

```
Task subclass: #LinkModules
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Task'



LinkModules methodsFor: standard

startExecution
   "ask for user attention"

   super startExecution.
   self runTools: #noUser.
   (DialogView confirm: 'Do you want to simulate Link
correct termination?')
      ifTrue: [self terminateExecution]
      ifFalse: [self editErrorReport]

LinkModules methodsFor: private

chooseTools: aSymbol
   "if necessary requires to the user for the tool he
prefers to perform the task"

   | collection inserted |
   inserted := Set new.
   collection := TripleArray new.
   output do: [:data | data provideTool isNil ifTrue:
         [| toolClass |
         (aSymbol = #noUser | (aSymbol = #auto)
            ifTrue: [Linker withAllSubclasses includes:
(toolClass := data class provideTool)]
            ifFalse: [(toolClass := data class provideTool)
notNil])
```

```
            ifTrue:
               [| chosen tmp |
               ((tmp := associatedTools instancesOfClass:
toolClass) includes: (chosen := responsible choiceOfClass:
toolClass))
                  ifTrue: [data creatorIs: chosen]
                  ifFalse: [tmp size > 1 ifFalse: [tmp size
isZero
                           ifTrue: [WarningWindow text: 'Any
tool of class ' , toolClass provideName , ' present.
Probable instantiation error!!']
                           ifFalse: [tmp do: [:tool | data
creatorIs: tool]]]
                        ifTrue: [(inserted includes:
toolClass)
                              ifFalse:
                                 [| temp |
                                 inserted add: toolClass.
                                 temp := DoubleArray new.
                                 tmp do: [:tool | temp add:
tool provideName with: tool].
                                 collection
                                    add: toolClass
provideName
                                    with: temp
                                    and: toolClass]]]]]].
   collection size isZero
      ifFalse:
         [aSymbol = #noUser ifTrue: [^false].
         DoubleListSelection
            on: collection
            for: self
            windowTitle: 'Tool Choice'
            firstListLabel: 'Tool Category'
            secondListLabel: 'AvailableTools'.
         self signal.
         userAction wait.
```

```
         self wait.
         inserted do: [:test | ^(responsible choiceOfClass:
test) notNil]].
   ^true

editErrorReport
   "ask for user attention and run editor"

   self askAttentionFor: 'Report errors for task ' , self
getName description: 'Edit error report of file ' , self
outputName.
   self runTools: #userInteracting.
   self openTerminationInterface

openTerminationInterface
   "opens an interface by which the user can specify
termination type"

   NotFailableTermination of: 'Error report of ' , self
outputName inTask: self

runTools: aSymbol
   "run appropriate tools on input and output data"

   | parameter |
   parameter := aSymbol.
   [self chooseTools: parameter]
      whileFalse:
         [self askAttentionFor: 'Choose tools for ' , self
getName description: 'Make tool choice and go on performing
activity'.
         aSymbol = #noUser ifTrue: [parameter := #auto]].
   (Array with: input with: output)
      do: [:receiver | receiver do:
            [:data |
            | tool |
            (tool := data provideTool) notNil ifTrue:
```

```
[aSymbol ~= #userInteracting & (Linker withAllSubclasses
includes: tool class) | (aSymbol = #userInteracting &
(Linker withAllSubclasses includes: tool class) not) ifTrue:
[tool runOn: data]]]]

LinkModules methodsFor: communication

getName
   "return a neme identifying the task"

   ^'Link ' , self outputName

suspend
   self critical: [self editErrorReport]

terminate
   self critical: [self giveFeedback]


LinkModules class
   instanceVariableNames: ''



LinkModules class methodsFor: initialization

initialize
   Predecessors new: self.
   Predecessors of: self is: Compile.
   WaitingFeedback new: self.
   WaitingFeedback of: self is: Edit.
   ResponsibleRole of: self is: Programmer.
   Output new: self; of: self is: (ExecutableFile
addProductor: self); of: self is: FeedbackDocument
```

```
Task subclass: #DevelopProgram
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Task'



DevelopProgram methodsFor: initialization

initialize: setOfRoles tools: toolsCollection responsible:
resp
   "initializes instance variables"

   super initialize: nil.
   assignedRoles := setOfRoles.
   associatedTools := toolsCollection.
   responsible := resp.
   output := InstanceConnection new

DevelopProgram methodsFor: special

requirements: req
   "create Configuration and connect it to the
RequirementDocument"

   self
      critical:
         [Output of: self class do:
            [:outputClass |
            | temp |
            temp := outputClass new.
            Input of: self class do:
               [:inputClass |
               | tmp |
               tmp := inputClass new.
```

```
                tmp is: req.
                tmp creatorIs: (associatedTools
instanceOfClass: inputClass provideTool).
                temp include: tmp].
             output add: temp].
          self changeState: #waitingPreconditions]


DevelopProgram class
   instanceVariableNames: ''



DevelopProgram class methodsFor: initialization

initialize
   "defines children classes"

   Children new: self.
   Children
      of: self
      is: AssignTasks
      numberOfInstances: 0.
   Children
      of: self
      is: SelectTool
      numberOfInstances: 1.
   Children
      of: self
      is: Design
      numberOfInstances: 1.
   Children
      of: self
      is: ReviewDesign
      numberOfInstances: 1.
   Children
      of: self
```

```
      is: Program
      numberOfInstances: 1.
   ResponsibleRole of: self is: ProjectManager.
   Output new: self.
   Output of: self is: (Configuration addProductor: self).
   Input new: self.
   Input of: self is: (RequirementDocument addProductor:
self)

DevelopProgram class methodsFor: instance creation

new: setOfRoles tools: toolConnection responsible: resp
   "creates a new instance of DevelopProgram"

   | newInstance |
   newInstance := super new.
   newInstance
      initialize: setOfRoles
      tools: toolConnection
      responsible: resp.
   ^newInstance
```

```
Task subclass: #SelectTool
   instanceVariableNames: 'win '
   classVariableNames: 'ToSelect '
   poolDictionaries: ''
   category: 'Task'



SelectTool methodsFor: comunication

selectionIs: selection
   "the ToolSelection interface returns the selection"

   self critical: [selection notNil
         ifTrue:
            [selection keysAndValuesDo: [:toolClass
:selectedTool | associatedTools removeAll: (associatedTools
instancesOfClass: toolClass); add: selectedTool].
            self terminateExecution]
         ifFalse: [self startExecution]]

SelectTool methodsFor: private

makeToolsCollection
   "returns a TripleArray whose first list is a list of
names of tool's classes; the
   second list is a list of DoubleArrays; the third list is
a list of tool's classes'
   identifiers.
   The DoubleArrays have a first list containing tools'
names and a second list
   containing tools' identifiers"

   | collection |
   collection := TripleArray new.
   ToSelect do:
```

```
      [:toolClass |
      | tmp |
      (tmp := associatedTools instancesOfClass: toolClass)
size > 1
         ifTrue:
            [| temp |
            temp := DoubleArray new.
            tmp do: [:tool | temp add: tool provideName
with: tool].
            collection
               add: toolClass provideName
               with: temp
               and: toolClass]].
   ^collection

openSelectionInterface
   "opens selection interface"

   | toolColl |
   toolColl := self makeToolsCollection.
   toolColl size isZero ifFalse: [DoubleListSelection
         on: toolColl
         for: self
         windowTitle: 'Tool Selection'
         firstListLabel: 'Tool Category'
         secondListLabel: 'AvailableTools']

SelectTool methodsFor: standard

startExecution
   "ask for user attention"

   super startExecution.
   self makeToolsCollection size isZero
      ifTrue:
         [self terminateExecution.
         ^self].
```

```
   self askAttentionFor: 'Select a tool' description:
'Select a tool for each class of tools in order to enforce
its use'.
   self openSelectionInterface

SelectTool methodsFor: relation handling

associatedTools: toolInstanceRelation
   "connects the new task to its associated tools"

   self critical: [associatedTools := toolInstanceRelation]


SelectTool class
   instanceVariableNames: ''



SelectTool class methodsFor: initialization

initialize
   ToSelect := InstanceConnection new.   "ToSelect add:
Linker.
   ToSelect add: Editor."
   ToSelect add: DesignTool
```

C.2 La categoria `Role`

```
Controlled subclass: #Role
   instanceVariableNames: 'assignedPersons '
   classVariableNames: 'Name '
   poolDictionaries: ''
   category: 'Role'



Role methodsFor: initialization

initialize: setOfPersons
   "connects the new instance to a set of persons playing
the role"

   assignedPersons := setOfPersons

Role methodsFor: accessing

players
   "returns a Set containing Persons playing the role"

   self critical: [^assignedPersons copy]

players: anInstanceRelation
   "connects players"

   self critical: [assignedPersons := anInstanceRelation]


Role class
   instanceVariableNames: ''
```

```
Role class methodsFor: instance creation

new: setOfPersons
   "creates a new instance of Role"

   ^super new initialize: setOfPersons

Role class methodsFor: comunication

provideName
   "return class's Name"

   ^Name at: self

Role class methodsFor: initialization

initialize
   "initialize class variables"

   Name := Dictionary new.
   self allSubclasses do: [:subclass | subclass setName]

setName
   "set the Name"

   Name at: self put: self printString
```

```
Role subclass: #Designer
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Role'
```

```
Role subclass: #ProjectManager
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Role'
```

```
Role subclass: #DesignReviewer
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Role'
```

```
Role subclass: #Programmer
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Role'
```

C.3 La categoria `Data`

```
Controlled subclass: #Data
   instanceVariableNames: 'isInput productors
faultsDocuments '
   classVariableNames: 'IsInput Productors '
   poolDictionaries: ''
   category: 'Data'



Data methodsFor: initialization

initialize

   productors := InstanceConnection new.
   faultsDocuments := InstanceConnection new.

Data methodsFor: relation handling

conditioningDataFrom: sender
   "return a Set containing all Data objects that somehow
condition self."

   self subclassResponsibility

conditioningTasks
   "return a Set containing all tasks that produce as output
data that somehow condition self."

   | collection |
   collection := Set new.
   (self conditioningDataFrom: self) do: [ :obj | | tmp |
      (tmp := obj productors) size isZero
      ifTrue: [
         ^nil
```

```
         ].
      collection addAll: tmp.
      ].
   ^collection

disconnect: aData
   "disconnect faults file aData"

   self critical: [
      faultsDocuments remove: aData.
      ].

faultsIn: aFeedbackDocument
   "connect  to FeedbackDocument containing faults"

   self critical: [
      faultsDocuments add: aFeedbackDocument.
      ].

predecessorTasksOfClass: taskClass
   "return a Set containing all tasks of class taskClass
that produce as output data that somehow condition self.
If a related data has no productor, it returns an empty
Set."

   | tmp |
   (self class productors includes: taskClass)
   ifFalse: [
      ^ Set new.
      ].
   self critical: [
      (tmp := productors instancesOfClass: taskClass)
isEmpty
      ifTrue: [
         ^nil
         ].
      ].
```

```
   ^tmp

productorIs: aTask
   "store task that produces self as output"

   self critical: [
      productors add: aTask.
      ].

productors

   | tmp |
   self critical: [
      tmp := productors copy.
      ].
   ^ tmp

provideName

   ^ self printString.

provideTool
   "return nil for objects that are not instances of
subclasses of SoftwareObject"

   ^nil

relatedDataFrom: sender
   "return a Set containing all Data objects that are
somehow needed to produce self."

   |tmp|
   tmp := self conditioningDataFrom: self.
   self critical: [
      tmp addAll: faultsDocuments.
      ].
   ^tmp
```

```
relatedDataIs: aData
   "connect all Data objects that somehow are related with
self."

   self subclassResponsibility


Data class
   instanceVariableNames: ''



Data class methodsFor: instance creation

new
   "ceate a new instance and invoke initialization method"

   ^(super new) initialize.

Data class methodsFor: initialization

initialize

   Productors := ClassRelation new.
   SoftwareObject initialize.

Data class methodsFor: relation handling

addProductor: taskClass
   "add a productor class"

   (Productors numberFor: self) isZero
   ifTrue: [
      Productors new: self
      ].
   Productors of: self is: taskClass.
```

```
    ^ self

productors
    "return productor classes"

    ^ (Productors of: self) copy

provideTool
    "return nil for classes that are not subclasses of
SoftwareObject"

    ^nil
```

```
Data subclass: #Configuration
   instanceVariableNames: 'components '
   classVariableNames: 'Components '
   poolDictionaries: ''
   category: 'Data'



Configuration methodsFor: testing

includes: SWObjClass
   "return true if a SW object of the specified class is
included "

   | tmp |
   self critical: [
      tmp := (components instancesOfClass: SWObjClass)
isEmpty not.
      ].
   ^ tmp

Configuration methodsFor: accessing

componentsOfClass: compClass
   "return the set of components of class compClass"

   | tmp |
   self critical: [
      tmp := components instancesOfClass: compClass.
      ].
   ^ tmp

Configuration methodsFor: management

include: newComponent
   "include a new component"
```

```
   self critical: [
      components add: newComponent.
      ].

Configuration methodsFor: initialization

initialize
   "creates new instances of connections"

   super initialize.
   components := InstanceConnection new.

Configuration methodsFor: relation handling

conditioningDataFrom: sender
   "return a Set containing all Data objects that are
somehow condition self."

   | tmp temp |
   self critical: [
      temp := components copy.
      ].
   tmp := Set new.
   temp do: [ :obj |
      obj = sender
      ifFalse: [
         tmp add: obj.
         ].
      ].
   ^tmp

provideName
   ^'Product configuration'
```

```
Data subclass: #SoftwareObject
   instanceVariableNames: 'creator file '
   classVariableNames: 'Creator '
   poolDictionaries: ''
   category: 'Data'



SoftwareObject methodsFor: initialization

is: associatedFile
   "store name of associated file"

   self critical: [
      file := associatedFile.
      self initialize.
      ].

SoftwareObject methodsFor: relation handling

creatorIs: tool
   "store the software object creator tool"

   self critical: [
      creator isNil & ((Creator of: self class) = tool
class)
      ifTrue: [
         creator := tool.
         ^true
         ].
      ].
   ^false

provideName

   self critical: [
```

```
      ^file.
      ].

provideTool
   "return the software object creator tool"

   self critical: [
      ^ creator.
      ].


SoftwareObject class
   instanceVariableNames: ''



SoftwareObject class methodsFor: initialization

initialize
   "initialize class variables"

   Creator := SingleClassRelation new.
   self allSubclasses do: [ :subclass |
      subclass initialize.
      ].

SoftwareObject class methodsFor: relation handling

provideTool
   "return the tool class operating on the software object"

   ^ Creator of: self.
```

```
SoftwareObject subclass: #TemporarySwObject
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Data'



TemporarySwObject methodsFor: removing

remove
   "remove instance"
```

```
TemporarySwObject subclass: #FeedbackDocument
   instanceVariableNames: 'referredTo '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Data'



FeedbackDocument methodsFor: initialization

initialize

   super initialize.
   referredTo := InstanceConnection new.

FeedbackDocument methodsFor: relation handling

conditioningDataFrom: sender
   "return a Set containing all Data objects that somehow
are related with self."

   ^Set new.

provideName

   self critical: [
      ^'Feedback document produced ',file.
      ].

relatedDataFrom: sender
   "return a Set containing all Data objects that somehow
are related with self."

   ^Set new.

relatedDataIs: aData
```

```
   "connect all Data objects that somehow are related with
self."

   self critical: [
      referredTo add: aData.
      ].
   aData faultsIn: self.

FeedbackDocument methodsFor: removing

remove
   "remove instance"

   | tmp |
   self critical: [
      tmp := referredTo copy.
      referredTo := InstanceConnection new.
      ].
   tmp do: [ :aData |
      aData disconnect: self.
      ].


FeedbackDocument class
   instanceVariableNames: ''



FeedbackDocument class methodsFor: initialization

initialize
   "initialize class connections"

   Creator of: self is: Editor.
```

```
SoftwareObject subclass: #ConfigurationComponent
   instanceVariableNames: 'owner '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Data'
```

```
ConfigurationComponent subclass: #SourceFile
   instanceVariableNames: 'describedBy use commentedBy '
   classVariableNames: 'DescribedBy Use '
   poolDictionaries: ''
   category: 'Data'



SourceFile methodsFor: initialization

initialize

   super initialize.
   describedBy := InstanceConnection new.
   use := InstanceConnection new.

SourceFile methodsFor: relation handling

commentedBy: aSWObj

   self critical: [
      commentedBy := aSWObj.
      ].

conditioningDataFrom: sender
   "return a Set containing all Data objects that somehow
condition self."

   | tmp temp |
   tmp := Set new.
   self critical: [temp := describedBy].
   temp do: [:obj | obj = sender
         ifFalse:
            [tmp addAll: (obj conditioningDataFrom: self).
            tmp add: obj]].
   self critical: [temp := commentedBy].
```

```
   temp notNil & (temp ~= sender)
      ifTrue:
         [tmp addAll: (temp conditioningDataFrom: self).
         tmp add: temp].
   ^tmp

describedBy: aSWObj

   self critical: [
      describedBy add: aSWObj.
      ].
   aSWObj describe: self.

relatedDataFrom: sender
   "return a Set containing all Data objects that somehow
are related with self."

   | tmp temp |
   tmp := self conditioningDataFrom: sender.
   self critical: [temp := use].
   temp do: [:obj | obj = sender ifFalse: [tmp add: obj]].
   self critical: [temp := faultsDocuments].
   tmp addAll: temp.
   ^tmp

usedBy: aSWObj

   self critical: [
      use add: aSWObj.
      ].


SourceFile class
   instanceVariableNames: ''
```

```
SourceFile class methodsFor: initialization

initialize
   "initialize class connections"

   Creator of: self is: Editor.
```

```
SourceFile subclass: #CompilableSourceFile
   instanceVariableNames: 'objects '
   classVariableNames: 'Objects Used '
   poolDictionaries: ''
   category: 'Data'



CompilableSourceFile methodsFor: initialization

initialize

   super initialize.
   objects := InstanceConnection new.

CompilableSourceFile methodsFor: relation handling

conditioningDataFrom: sender
   "return a Set containing all Data objects that somehow
condition self."

   | tmp temp |
   tmp := super conditioningDataFrom: sender.
   self critical: [
      temp := use copy.
      ].
   temp do: [ :obj |
      obj = sender
      ifFalse: [
         tmp addAll: (obj conditioningDataFrom: self).
         tmp add: obj.
         ].
      ].
   ^tmp

generate: aSWObj
```

```
   self critical: [
      objects add: aSWObj.
      ].
   aSWObj generatedBy: self

use: aSWObj

   self critical: [
      use add: aSWObj.
      ].
   aSWObj usedBy: self.
```

```
ConfigurationComponent subclass: #DesignDocument
   instanceVariableNames: 'described toMeet '
   classVariableNames: 'Described '
   poolDictionaries: ''
   category: 'Data'



DesignDocument methodsFor: relation handling

conditioningDataFrom: sender
   "return a Set containing all Data objects that somehow
condition self."

   ^Set new

describe: aSWObj

   self critical: [
      described add: aSWObj.
      ].

meet: aSWObj

   self critical: [
      toMeet add: aSWObj.
      ].
   aSWObj metBy: self.

DesignDocument methodsFor: initialization

initialize

   super initialize.
   described := InstanceConnection new.
   toMeet := InstanceConnection new.
```

```
DesignDocument class
   instanceVariableNames: ''



DesignDocument class methodsFor: initialization

initialize
   "initialize class connections"

   Creator of: self is: DesignTool.
```

```
ConfigurationComponent subclass: #CommentFile
   instanceVariableNames: 'commented '
   classVariableNames: 'Commented '
   poolDictionaries: ''
   category: 'Data'



CommentFile methodsFor: relation handling

comment: aSWObj

   self critical: [
      commented := aSWObj.
      ].
   aSWObj commentedBy: self.

conditioningDataFrom: sender
   "return a Set containing all Data objects that somehow
condition self."

   ^Set new.

relatedDataFrom: sender
   "return a Set containing all Data objects that somehow
are related to self."

   | tmp |
   tmp := super relatedDataFrom: sender.
   self critical: [
      commented notNil & (commented ~= sender)
      ifTrue: [
         tmp add: commented.
         ].
      ].
   ^tmp
```

```
CommentFile class
   instanceVariableNames: ''



CommentFile class methodsFor: initialization

initialize
   "initialize class connections"

   Creator of: self is: Editor.
```

```
ConfigurationComponent subclass: #ObjectFile
   instanceVariableNames: 'source executables '
   classVariableNames: 'Executables Source '
   poolDictionaries: ''
   category: 'Data'



ObjectFile methodsFor: relation handling

compose: aSWObj

   self critical: [
      executables add: aSWObj.
      ].
   aSWObj composedBy: self.

conditioningDataFrom: sender
   "return a Set containing all Data objects that somehow
condition self."

   | tmp temp |
   tmp := Set new.
   self critical: [
      temp := source.
      ].
   temp notNil & temp ~= sender
   ifTrue: [
      tmp addAll: (temp conditioningDataFrom: self).
      tmp add: temp.
      ].
   ^tmp

generatedBy: aSWObj

   self critical: [
```

```
      source := aSWObj.
      ].

ObjectFile methodsFor: initialization

initialize

   super initialize.
   executables := InstanceConnection new.


ObjectFile class
   instanceVariableNames: ''



ObjectFile class methodsFor: initialization

initialize
   "initialize class connections"

   Creator of: self is: CompileTool.
```

```
ConfigurationComponent subclass: #RequirementDocument
   instanceVariableNames: 'metBy '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Data'



RequirementDocument methodsFor: relation handling

conditioningDataFrom: sender
   "return a Set containing all Data objects that somehow
condition self."

   ^Set new.

metBy: aSWObj

   self critical: [
      metBy add: aSWObj.
      ].

RequirementDocument methodsFor: initialization

initialize

   super initialize.
   metBy := InstanceConnection new.


RequirementDocument class
   instanceVariableNames: ''



RequirementDocument class methodsFor: initialization
```

```
initialize
   "initialize class connections"

   Creator of: self is: Editor.
```

```
ConfigurationComponent subclass: #ExecutableFile
   instanceVariableNames: 'objects '
   classVariableNames: 'Objects '
   poolDictionaries: ''
   category: 'Data'



ExecutableFile methodsFor: relation handling

composedBy: aSWObj

   self critical: [
      objects add: aSWObj.
      ].

conditioningDataFrom: sender
   "return a Set containing all Data objects that somehow
condition self."

   | tmp temp |
   tmp := Set new.
   self critical: [
      temp := objects copy.
      ].
   temp do: [ :obj |
      obj = sender
      ifFalse: [
         tmp addAll: (obj conditioningDataFrom: self).
         tmp add: obj.
         ].
      ].
   ^tmp

ExecutableFile methodsFor: initialization
```

```
initialize

   super initialize.
   objects := InstanceConnection new.


ExecutableFile class
   instanceVariableNames: ''



ExecutableFile class methodsFor: initialization

initialize
   "initialize class connections"

   Creator of: self is: Linker.
```

C.4 La categoria `Tool`

```
Controlled subclass: #Tool
   instanceVariableNames: 'name '
   classVariableNames: 'Name '
   poolDictionaries: ''
   category: 'Tool'



Tool methodsFor: standard

runOn: data
   "run self on data. This is a simulation and then a
warning is shown"

   | tmp |
   self critical: [tmp := name copy].
   WarningWindow text: 'Opened ' , tmp , ' on ' , data
provideName

Tool methodsFor: communication

provideName
   "comment stating purpose of message"

   self critical: [^name]

Tool methodsFor: initialization

initialize: aName
   "stores Tool's name"

   name := aName
```

```
Tool class
   instanceVariableNames: ''



Tool class methodsFor: instance creation

new: toolName
   "creates a new instance of a tool and stores its name"

   ^super new initialize: toolName

Tool class methodsFor: initialization

initialize
   "comment stating purpose of message"

   Name := Dictionary new

initialize: className
   "comment stating purpose of message"

   Name at: self put: className

Tool class methodsFor: comunication

provideName
   "comment stating purpose of message"

   ^Name at: self
```

```
Tool subclass: #Linker
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Tool'
```

```
Tool subclass: #Editor
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Tool'
```

```
Tool subclass: #DesignTool
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Tool'
```

```
Tool subclass: #CompileTool
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Tool'
```

C.5 La categoria `Persons`

```
Controlled subclass: #Person
   instanceVariableNames: 'chosenTools agenda name '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Persons'



Person methodsFor: comunication

choiceIs: tool
   "record chosen tool of class toolClass"

   | tmp |
   self critical: [tmp := chosenTools add: tool].
   ^tmp

choiceOfClass: toolClass
   "return chosen tool of class toolClass"

   | tmp |
   self critical: [tmp := chosenTools instanceOfClass:
toolClass].
   ^tmp

provideName
   "return name of the person"

   | tmp |
   self critical: [tmp := name].
   ^tmp

recordActivity: activityName from: taskId description: descr
```

```
   "request the Agenda to record a new activity received
from Task taskId for
   aMatter"

   [agenda
      record: activityName
      from: taskId
      description: descr] fork

Person methodsFor: initialization

initialize: personName
   "stores person's name"

   name := personName.
   agenda := Agenda ofPersonNamed: name.
   chosenTools := InstanceConnection new


Person class
   instanceVariableNames: ''



Person class methodsFor: instance creation

new: personName
   "creates a new instance and stores perso's name"

   | newPerson |
   newPerson := super new.
   ^newPerson initialize: personName
```

C.6 La categoria Relations

```
Dictionary variableSubclass: #BooleanInstanceRelation
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Relations'



BooleanInstanceRelation methodsFor: adding

add: objectId
   "connects a new object"

   self at: objectId put: false

addTrue: childId
   "connects a new task satisfying the condition to the
task"

   self at: childId put: true

BooleanInstanceRelation methodsFor: accessing

instanceOfClass: classId
   "returns the related instances of specified class"

   self keysDo: [:child | child class == classId ifTrue:
[^child]].
   ^nil

instancesOfClass: classId
   "returns the related instances of specified class"

   | temp |
```

```
   temp := Array new.
   self keysDo: [:child | child class == classId ifTrue:
[temp add: child]].
   ^temp

BooleanInstanceRelation methodsFor: testing

allCompleted
   "returns True if all related Tasks are completed"

   self do: [:completed | completed ifFalse: [^false]].
   ^true

allCompleted: taskSet
   "return True if all tasks of taskSet class are completed
or are not children"

   taskSet do: [:task | (self at: task ifAbsent: [true])
         ifFalse: [^false]].
   ^true

allCompletedOfClass: taskClass
   "return True if all Children of TaskClass class are
completed"

   self keysAndValuesDo: [:child :completed | child class ==
taskClass & completed not ifTrue: [^false]].
   ^true

includes: connectedObject
   ^self includesKey: connectedObject

instancesNumberOfClass: childClass
   "returns the number of Child's instances of class
childClass"

   | count |
```

```
   count := 0.
   self keysDo: [:child | child class == childClass ifTrue:
[count := count + 1]].
   ^count
```

```
Dictionary variableSubclass: #ClassRelation
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Relations'



ClassRelation methodsFor: adding

new: relatingClass
   "create a new entry for relating class"

   self at: relatingClass put: Set new

of: relatingClass is: relatedClass
   "stores the related class"

   (self at: relatingClass)
      add: relatedClass

ClassRelation methodsFor: accessing

of: relatingClass
   "return a set containing related classes"

   ^self at: relatingClass ifAbsent: [Set new]

ClassRelation methodsFor: enumeration

numberFor: relatingClass
   "returns number of entries for relatingClass"

   (self includesKey: relatingClass)
      ifTrue: [^(self at: relatingClass) size].
   ^0
```

```
of: relatingClass do: block
   "executes block on every predecessor of relatingClass"

   (self includesKey: relatingClass)
      ifTrue: [^(self at: relatingClass)
            do: block]
```

```
ClassRelation variableSubclass: #SingleClassRelation
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Relations'



SingleClassRelation methodsFor: adding

of: relatingClass is: relatedClass
   "creates a new entry for relating class and connects it
to related class"

   self at: relatingClass put: relatedClass

SingleClassRelation methodsFor: enumeration

of: relatingClass
   "returns the related role if exits"

   ^self at: relatingClass ifAbsent: [^nil]
```

```
ClassRelation variableSubclass: #ChildrenClassRelation
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Relations'



ChildrenClassRelation methodsFor: adding

new: fatherClass
   "creates a new entry for the new father"

   self at: fatherClass put: Dictionary new

of: fatherClass is: childId numberOfInstances: inst
   "stores the child class and the number of instances
wanted for that task"

   (self at: fatherClass)
      at: childId put: inst

ChildrenClassRelation methodsFor: enumeration

childrenAndNumberOf: fatherClass do: block
   "executes block on each Child"

   ^(self at: fatherClass ifAbsent: [^nil])
      keysAndValuesDo: block

everyChildrenOf: fatherClass do: block
   "executes block on each Child"

   (self includesKey: fatherClass)
      ifTrue: [^(self at: fatherClass)
            keysDo: block]
```

```
Set variableSubclass: #InstanceConnection
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Relations'



InstanceConnection methodsFor: accessing

instanceOfClass: relatedClass
   "searches the instance of the relation connected with
relatedClass"

   (self instancesOfClass: relatedClass)
      do: [:temp | ^temp].
   ^nil

instancesOfClass: classOfRelated
   "returns the set of instances of class childClass"

   | tmp |
   tmp := Set new.
   self do: [:related | related class = classOfRelated
ifTrue: [tmp add: related]].
   ^tmp

InstanceConnection methodsFor: adding

add: elem
   super add: elem.
   ^self
```

```
InstanceConnection variableSubclass:
#UniqueInstanceConnection
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Relations'



UniqueInstanceConnection methodsFor: adding

add: anItem
   "add anItem mantaining instance unicity"

   | tmp |
   (tmp := self instanceOfClass: anItem class) notNil
ifTrue: [self remove: tmp].
   super add: anItem

UniqueInstanceConnection methodsFor: accessing

instanceOfClass: relatedClass
   "search the instance of the relation connected with
relatedClass"

   self do: [:temp | temp class == relatedClass ifTrue:
[^temp]].
   ^nil
```

C.7 La categoria `Utilities`

```
ScheduledWindow subclass: #WarningWindow
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Utilities'



WarningWindow methodsFor: initialize

initialize: aString
   | origin wrapper text extent |
   origin := Window currentOrigin - 50.
   text := aString asComposedText.
   text compositionWidth: 200.
   wrapper := BorderedWrapper on: text.
   wrapper borderWidth: 10; borderColor: self
backgroundColor.
   self component: wrapper.
   extent := wrapper preferredBounds extent.
   self label: 'Warning'.
   self openNoTerminateIn: (origin extent: extent)


WarningWindow class
   instanceVariableNames: ''



WarningWindow class methodsFor: instance creation

text: aString
   "create and display a warning window"
```

```
^super new initialize: aString
```

```
Object subclass: #Controlled
   instanceVariableNames: 'semaphore '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Utilities'



Controlled methodsFor: communication

critical: aBlock
   semaphore critical: aBlock

signal
   semaphore signal

wait
   semaphore wait

Controlled methodsFor: initialization

thisInitialize
   semaphore := Semaphore forMutualExclusion.
   ^self


Controlled class
   instanceVariableNames: ''



Controlled class methodsFor: instance creation

new
   "create a new instance"
```

```
^super new thisInitialize
```

```
BorderedWrapper subclass: #Button
   instanceVariableNames: 'buttonLabel action status
relatedObject button '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Utilities'



Button methodsFor: status control

disable
   "disable the button"

   status := #disabled.
   button beVisual: (buttonLabel asText
         emphasizeFrom: 1
         to: buttonLabel size
         with: #italic) asComposedText.
   self borderWidth: 1; inset: 2

enable
   "hilight the button"

   status := #enabled.
   self model beVisual: buttonLabel asText asComposedText
ifTrue: buttonLabel asText allBold asComposedText.
   self borderWidth: 3

pressed
   "take appropriate action if the button is active"

   status = #enabled ifTrue: action

Button methodsFor: positioning
```

```
centerX: center bottomFraction: bottom offset: offset
   "comment stating purpose of message"

   | bounds |
   bounds := self preferredBounds.
   self layout: ((LayoutFrame new) leftFraction: center
offset: 0 - (bounds width / 2); topFraction: bottom offset:
offset - bounds height; rightFraction: center offset: bounds
width / 2; bottomFraction: bottom offset: offset)

centerX: x centerY: y
   "comment stating purpose of message"

   | bounds |
   bounds := self preferredBounds.
   self layout: ((LayoutFrame new) leftFraction: x offset:
(bounds width / 2) negated; topFraction: y offset: (bounds
height / 2) negated; rightFraction: x offset: bounds width /
2; bottomFraction: y offset: bounds height / 2)

Button methodsFor: bounds

layout
   "return the layout"

   ^layout

Button methodsFor: initialization

initialize: aLabel action: block model: relatedModel
   "initialize variables"

   buttonLabel := aLabel.
   status := #enabled.
   action := block.
   relatedObject := relatedModel
```

```
initialize: aButton label: aLabel action: block model:
relatedModel
   "initialize variables"

   button := aButton.
   buttonLabel := aLabel.
   status := #enabled.
   action := block.
   relatedObject := relatedModel


Button class
   instanceVariableNames: ''



Button class methodsFor: instance creation

label: buttonLabel model: relatedModel execute: method
addTo: container
   "creates a new button"

   | button wrapper |
   button := LabeledBooleanView new.
   wrapper := (super on: button)
            borderWidth: 3.
   button beVisual: buttonLabel asText asComposedText
ifTrue: buttonLabel asText allBold asComposedText; model:
((PluggableAdaptor on: wrapper)
         getBlock: [:model | false]
         putBlock: [:model :value | model perform: #pressed]
         updateBlock: [:model :value :parameter | true]).
   button controller beTriggerOnUp.
   container addWrapper: wrapper.
   wrapper
      initialize: button
      label: buttonLabel
```

```
    action: [relatedModel perform: method]
    model: relatedModel.
^wrapper
```

```
BorderedWrapper subclass: #Label
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Utilities'




Label methodsFor: bounds

layout
   "return label layout"

   ^layout



Label class
   instanceVariableNames: ''




Label class methodsFor: instance creation

text: labelText addTo: container origin: point
rightFraction: right
   "creates a new label"

   | bounds wrapper |
   wrapper := super on: labelText asComposedText.
   bounds := wrapper preferredBounds.
   wrapper layout: ((LayoutFrame new) leftFraction: point x;
topFraction: point y; rightFraction: right; bottomFraction:
point y offset: bounds height).
```

```
container addWrapper: wrapper.
^wrapper
```

```
Object subclass: #DoubleArray
   instanceVariableNames: 'firstElement secondElement '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Utilities'



DoubleArray methodsFor: adding

add: firstValue with: secondValue
   "adds an double element to the list"

   firstElement := firstElement copyWith: firstValue.
   secondElement := secondElement copyWith: secondValue

DoubleArray methodsFor: testing

size
   "return the number of elements"

   ^firstElement size

DoubleArray methodsFor: accessing

at: index put: value1 with: value2
   "changes the content of position index"

   firstElement size >= index
      ifTrue:
         [firstElement at: index put: value1.
         secondElement at: index put: value2.
         ^true].
   ^false

first
```

```
   "returns first list"

   ^firstElement

firstAt: index
   "returns an element of the first list"

   ^firstElement at: index

second
   "returns second list"

   ^secondElement

secondAt: index
   "returns an element of the second list"

   ^secondElement at: index

DoubleArray methodsFor: private

remove: position from: anArray
   "removes element in position from anArray"

   position isZero | (position > anArray size)
      ifFalse:
         [| tmp |
         tmp := Array new: anArray size - 1.
         1 to: position - 1 do: [:count | tmp at: count put:
(anArray at: count)].
         position to: anArray size - 1 do: [:count | tmp at:
count put: (anArray at: count + 1)].
         anArray become: tmp].
   ^anArray


DoubleArray methodsFor: removing
```

```
removePosition: index
   "removes the element pointed by index"

   index isZero | (index > firstElement size)
      ifFalse:
         [firstElement := self remove: index from:
firstElement.
         secondElement := self remove: index from:
secondElement]


DoubleArray methodsFor: initialization

initialize
   "initializes a new instance"

   firstElement := Array new.
   secondElement := Array new

DoubleArray methodsFor: enumerating

do: block
   "execute block for each element"

   ^firstElement with: secondElement do: block



DoubleArray class
   instanceVariableNames: ''



DoubleArray class methodsFor: instance creation

new
```

```
"creates a new instance and calls initialization method"

^super new initialize
```

```
DoubleArray subclass: #TripleArray
   instanceVariableNames: 'thirdElement '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Utilities'



TripleArray methodsFor: adding

add: firstValue with: secondValue and: thirdValue
   "adds an triple element to the list"

   super add: firstValue with: secondValue.
   thirdElement := thirdElement copyWith: thirdValue

TripleArray methodsFor: removing

removePosition: index
   "removes the element pointed by index"

   index isZero | (index > firstElement size)
      ifFalse:
         [super removePosition: index.
         thirdElement := self remove: index from:
thirdElement]

TripleArray methodsFor: initialization

initialize
   "initializes a new instance"

   super initialize.
   thirdElement := Array new

TripleArray methodsFor: accessing
```

```
at: index put: value1 with: value2 and: value3
   "replace values at position index"

   (self
      at: index
      put: value1
      with: value2)
      ifTrue:
         [thirdElement at: index put: value3.
         ^true].
   ^false

third
   "returns third list"

   ^thirdElement

thirdAt: index
   "returns the index element of third list"

   ^thirdElement at: index

TripleArray methodsFor: enumerating

do: block
   "execute block for each element"

   ^1 to: firstElement size do: [:index | block
         value: (firstElement at: index)
         value: (secondElement at: index)
         value: (thirdElement at: index)]
```

C.8 La categoria `User interfaces`

```
Model subclass: #Agenda
   instanceVariableNames: 'selection list relatedPerson
semaphore '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'User interfaces'



Agenda methodsFor: list menu

activityListMenu
   "provides a menu for the Agenda selection list"

   selection isZero
      ifTrue: [^nil]
      ifFalse: [^PopUpMenu labelList: #(#('show description'
#execute ) ) values: #(#showComment #gotAttention )]

gotAttention
   "notify task that user is ready"

   | receiver |
   semaphore
      critical:
         [receiver := (list secondAt: selection)
                  at: 2.
         list removePosition: selection].
   receiver gotUserAttention.
   self changed: #agenda

showComment
   "show comment for the selected item"
```

```
   | commentWindow item description wrapper extent view |
   semaphore
      critical:
         [item := list firstAt: selection.
         description := (list secondAt: selection)
                  at: 1].
   commentWindow := ScheduledWindow new.
   view := description asComposedText.
   view compositionWidth: 300.
   wrapper := BorderedWrapper on: view.
   wrapper borderWidth: 10; borderColor: commentWindow
backgroundColor.
   extent := wrapper preferredBounds extent.
   commentWindow label: 'Description of activity: ' , item.
   commentWindow component: wrapper.
   commentWindow openNoTerminateIn: (Window currentOrigin +
50 extent: extent)

Agenda methodsFor: comunication

record: elementName from: taskId description: descr
   "records a new activity received from Task taskId for
aMatter"

   | temp |
   temp := Array with: descr with: taskId.
   semaphore critical: [list add: elementName with: temp].
   self changed: #agenda

Agenda methodsFor: handling

activityList
   "gives to the agenda the list of activities"

   | tmp |
   semaphore critical: [tmp := list first].
   ^tmp
```

```
selectionIs: index
   "store new selection"

   semaphore critical: [selection := index]

Agenda methodsFor: initialization

displayWithTitle: title
   "creates the window"

   | agendaView container agendaWindow |
   container := CompositePart new.
   agendaWindow := ScheduledWindow new.
   agendaWindow label: title.
   agendaView := SelectionInListView
            noDelimitersOn: self
            aspect: #agenda
            change: #selectionIs:
            list: #activityList
            menu: #activityListMenu
            initialSelection: nil.
   container add: (LookPreferences edgeDecorator on:
agendaView)
      borderedIn: (0 @ 0 extent: 1 @ 1).
   agendaWindow component: container.
   agendaWindow controller: NoCloseController new.
   agendaWindow openNoTerminateIn: (Window currentOrigin +
50 extent: 200 @ 200)

personNameIs: personName
   "initialize the Agenda"

   list := DoubleArray new.
   semaphore := Semaphore forMutualExclusion.
   selection := 0.
   self displayWithTitle: 'Agenda of ' , personName
```

```
Agenda class
   instanceVariableNames: ''



Agenda class methodsFor: instance creation

ofPersonNamed: personName
   "creates a new instance of Agenda"

   ^super new personNameIs: personName
```

```
Model subclass: #DoubleListSelection
   instanceVariableNames: 'firstListSelection globalList
selectedItems selectionWindow connectedTask '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'User interfaces'



DoubleListSelection methodsFor: initialization

displayInterface: title firstListLabel: label1
secondListLabel: label2
   "create the interface for the choice"

   | firstListView secondListView container button listLabel
|
   container := CompositePart new.
   selectionWindow := ScheduledWindow new.
   selectionWindow label: title.   "end button"
   (Button
      label: ' End '
      model: self
      execute: #endPressed
      addTo: container)
      centerX: 0.25
      bottomFraction: 1
      offset: -5.   "cancel button"
   button := (Button
            label: ' Cancel '
            model: self
            execute: #cancelPressed
            addTo: container)
            centerX: 0.75
            bottomFraction: 1
            offset: -5.   "titles"
```

```
   listLabel := Label
            text: label1
            addTo: container
            origin: 0 @ 0
            rightFraction: 0.4.
   Label
      text: label2
      addTo: container
      origin: 0.4 @ 0
      rightFraction: 1.   "first selection list"
   firstListView := SelectionInListView
            noDelimitersOn: self
            aspect: #firstList
            change: #firstListSelectionIs:
            list: #firstList
            menu: nil
            initialSelection: nil.
   container add: (LookPreferences edgeDecorator on:
firstListView) noMenuBar borderedIn: ((LayoutFrame new)
leftFraction: 0 offset: 0; rightFraction: 0.4 offset: 0;
topFraction: listLabel layout bottomFraction offset:
listLabel layout bottomOffset; bottomFraction: button layout
topFraction offset: button layout topOffset - 3).   "second
selection list"
   secondListView := SelectionInListView
            noDelimitersOn: self
            aspect: #secondList
            change: #secondListSelectionIs:
            list: #secondList
            menu: nil
            initialSelection: #initialItem.
   container add: (LookPreferences edgeDecorator on:
secondListView) noMenuBar borderedIn: ((LayoutFrame new)
leftFraction: 0.4 offset: 0; rightFraction: 1 offset: 0;
topFraction: listLabel layout bottomFraction offset:
listLabel layout bottomOffset; bottomFraction: button layout
topFraction offset: button layout topOffset - 3).
```

```
"displays the window"
   selectionWindow component: container.
   selectionWindow controller: NoCloseController new.
   selectionWindow openNoTerminateIn: (Window currentOrigin
+ 50 extent: 400 @ 200)

selectFrom: choiceCollection for: aTask windowTitle: title
firstListLabel: label1 secondListLabel: label2
   "choiceCollection is a TripleArray whose first list is a
list of names that go
   into first selection list; the second list is a list of
DoubleArrays; the third list
   is a list of items' identifiers.
   The DoubleArrays have a first list containing second
selection list items'
   names and a second list containing items' identifiers"

   connectedTask := aTask.
   globalList := DoubleArray new.
   selectedItems := Dictionary new.
   firstListSelection := 0.
   1 to: choiceCollection size do:
      [:firstIndex |
      | tmp secondListArray |
      secondListArray := choiceCollection secondAt:
firstIndex.
      tmp := DoubleArray new.
      selectedItems at: (choiceCollection thirdAt:
firstIndex)
         put: nil.
      1 to: secondListArray size do: [:secondIndex | tmp
add: (secondListArray firstAt: secondIndex)
            with: (secondListArray secondAt: secondIndex)].
      globalList add: (choiceCollection firstAt: firstIndex)
         with: (Array with: (choiceCollection thirdAt:
firstIndex)
               with: tmp)].
```

```
   self
      displayInterface: title
      firstListLabel: label1
      secondListLabel: label2

DoubleListSelection methodsFor: handling

cancelPressed
   "action taken because the cancel button is pressed"

   [connectedTask selectionIs: nil] fork.
   selectionWindow controller close

endPressed
   "action taken because the end button is pressed"

   (selectedItems includes: nil)
      ifFalse:
         [| temp |
         temp := Dictionary new.
         selectedItems keysAndValuesDo: [:firstListItem
:selection | temp at: firstListItem put: (selection at: 1)].
         [connectedTask selectionIs: temp] fork.
         selectionWindow controller close]
      ifTrue: [WarningWindow text: 'Select one item in the
second list for each item in the first list']

firstList
   "gives to the DoubleListSelection interface the first
list"

   | temp |
   temp := globalList first.
   ^temp

firstListSelectionIs: index
   "record first list selection"
```

```
   firstListSelection := index.
   self changed: #secondList

initialItem
   "return the initial tool selection"

   | selectedFirstListItem secondListItem |
   firstListSelection isZero
      ifFalse:
         [selectedFirstListItem := (globalList secondAt:
firstListSelection)
                  at: 1.
         secondListItem := selectedItems at:
selectedFirstListItem.
         secondListItem notNil
            ifTrue: [^secondListItem at: 2]
            ifFalse: [^0]].
   ^0

secondList
   "gives to the DoubleListSelection interface the list of
items of selected first
   list element"

   firstListSelection isZero ifFalse: [^((globalList
secondAt: firstListSelection)
         at: 2) first].
   ^nil

secondListSelectionIs: index
   "store selected item"

   | selectedSecondListItem selectedFirstListItem |
   index isZero
      ifFalse:
         [selectedFirstListItem := (globalList secondAt:
```

```
firstListSelection)
                  at: 1.
         selectedSecondListItem := ((globalList secondAt:
firstListSelection)
                  at: 2)
                  secondAt: index.
         selectedItems at: selectedFirstListItem put: (Array
with: selectedSecondListItem with: index)]
      ifTrue: [firstListSelection isZero
            ifFalse:
               [selectedFirstListItem := (globalList
secondAt: firstListSelection)
                        at: 1.
               selectedItems at: selectedFirstListItem put:
nil]]


DoubleListSelection class
   instanceVariableNames: ''



DoubleListSelection class methodsFor: instance creation

on: list for: aTask windowTitle: title firstListLabel:
label1 secondListLabel: label2
   "creates a new instance of DoubleSelection interface for
the task aTask"

   ^super new
      selectFrom: list
      for: aTask
      windowTitle: title
      firstListLabel: label1
      secondListLabel: label2
```

```
DoubleListSelection subclass: #MultipleListMultipleSelection
   instanceVariableNames: 'secondListASelection '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'User interfaces'



MultipleListMultipleSelection methodsFor: initialization

displayInterface: title labels: labelsArray
   "create the interface for the choice"

   | listView container button label |
   container := CompositePart new.
   selectionWindow := ScheduledWindow new.
   selectionWindow label: title.   "end button"
   (Button
      label: ' End '
      model: self
      execute: #endPressed
      addTo: container)
      centerX: 0.15
      bottomFraction: 1
      offset: -5.   "cancel button"
   button := (Button
            label: ' Cancel '
            model: self
            execute: #cancelPressed
            addTo: container)
            centerX: 0.45
            bottomFraction: 1
            offset: -5.   "first selection list"
   label := Label
            text: (labelsArray at: 1)
            addTo: container
```

```
            origin: 0 @ 0
            rightFraction: 0.3.
   listView := SelectionInListView
            noDelimitersOn: self
            aspect: #firstList
            change: #firstListSelectionIs:
            list: #firstList
            menu: nil
            initialSelection: nil.
   container add: (LookPreferences edgeDecorator on:
listView) noMenuBar borderedIn: ((LayoutFrame new)
leftFraction: 0; rightFraction: 0.3; topFraction: label
layout bottomFraction offset: label layout bottomOffset;
bottomFraction: button layout topFraction offset: button
layout topOffset - 3).   "second selection list A"
   label := Label
            text: (labelsArray at: 2)
            addTo: container
            origin: 0.3 @ 0
            rightFraction: 0.6.
   listView := SelectionInListView
            noDelimitersOn: self
            aspect: #secondList
            change: #secondListASelectionIs:
            list: #secondListA
            menu: nil
            initialSelection: #initialItem2A.
   container add: (LookPreferences edgeDecorator on:
listView) noMenuBar borderedIn: ((LayoutFrame new)
leftFraction: 0.3; rightFraction: 0.6; topFraction: label
layout bottomFraction offset: label layout bottomOffset;
bottomFraction: 0.7 offset: -2).   "second selection list B"
   label := Label
            text: (labelsArray at: 3)
            addTo: container
            origin: 0.3 @ 0.7
            rightFraction: 0.6.
```

```
   listView := SelectionInListView
            noDelimitersOn: self
            aspect: #secondList
            change: #secondListBSelectionIs:
            list: #secondListB
            menu: nil
            initialSelection: #initialItem2B.
   container add: (LookPreferences edgeDecorator on:
listView) noMenuBar noVerticalScrollBar borderedIn: ((label
layout copy) topFraction: label layout bottomFraction
offset: label layout bottomOffset; bottomFraction: label
layout bottomFraction offset: label layout bottomOffset * 2
- label layout topOffset).   "third selection list A"
   label := Label
            text: (labelsArray at: 4)
            addTo: container
            origin: 0.6 @ 0
            rightFraction: 1.
   listView := SelectionInListView
            noDelimitersOn: self
            aspect: #thirdListAB
            change: #thirdListASelectionIs:
            list: #thirdListA
            menu: nil
            initialSelection: nil.
   container add: (LookPreferences edgeDecorator on:
listView) noMenuBar borderedIn: ((LayoutFrame new)
leftFraction: 0.6; rightFraction: 1; topFraction: label
layout bottomFraction offset: label layout bottomOffset;
bottomFraction: 0.35 offset: -2).   "third selection list B"
   label := Label
            text: (labelsArray at: 5)
            addTo: container
            origin: 0.6 @ 0.35
            rightFraction: 1.
   listView := SelectionInListView
            noDelimitersOn: self
```

```
            aspect: #thirdListAB
            change: #thirdListBSelectionIs:
            list: #thirdListB
            menu: nil
            initialSelection: nil.
   container add: (LookPreferences edgeDecorator on:
listView) noMenuBar borderedIn: ((LayoutFrame new)
leftFraction: 0.6; rightFraction: 1; topFraction: label
layout bottomFraction offset: label layout bottomOffset;
bottomFraction: 0.7 offset: -2).   "third selection list C"
   label := Label
            text: (labelsArray at: 6)
            addTo: container
            origin: 0.6 @ 0.7
            rightFraction: 1.
   listView := SelectionInListView
            noDelimitersOn: self
            aspect: #thirdListC
            change: #thirdListCSelectionIs:
            list: #thirdListC
            menu: nil
            initialSelection: #initialItem3C.
   container add: (LookPreferences edgeDecorator on:
listView) noMenuBar borderedIn: ((LayoutFrame new)
leftFraction: 0.6; rightFraction: 1; topFraction: label
layout bottomFraction offset: label layout bottomOffset;
bottomFraction: 1).
   selectionWindow controller: NoCloseController new.
   selectionWindow component: container.
   selectionWindow openNoTerminateIn: (Window currentOrigin
+ 50 extent: 400 @ 400)

selectFrom: choiceCollection for: aTask windowTitle: title
labels: labelsArray
   "initialize the varibles.
   globalList is derived from choiceCollection and has the
following structure:
```

```
   collection is a TripleArray. Each element is:
   a task name;       (A);                  (B)
   (A) is a TripleArray. Each element is:
   a role name;      (C) ;                  a role
identifier
   (B) is a 4 slots Array. Each slot contains:
   a task identifier;   a responsible role name   (D)

selected slot of (D)
   (C) is a 2 slots Array. The first slot contains:
   a DoubleArray whose elements are:
   a person name   a person identifier {available to play
related role}
   a DoubleArray whose elements are:
   a person name   a person identifier {selected to play
related role}
   (D) is a DoubleArray. Each element id:
   a person name   a person identifier {playing responsible
role}
   "

   connectedTask := aTask.
   globalList := choiceCollection.
   secondListASelection := Dictionary new.
   firstListSelection := 0.
   1 to: choiceCollection size do:
      [:firstIndex |
      | tripleArray2 |
      secondListASelection at: firstIndex put: 0.
      tripleArray2 := choiceCollection secondAt: firstIndex.
      1 to: tripleArray2 size do:
         [:secondIndex |
         | doubleArray firstElement thirdElement |
         firstElement := tripleArray2 firstAt: secondIndex.
         doubleArray := tripleArray2 secondAt: secondIndex.
         thirdElement := tripleArray2 thirdAt: secondIndex.
```

```
"list of available and list of selected"
         tripleArray2
            at: secondIndex
            put: firstElement
            with: (Array with: doubleArray with: DoubleArray
new)
            and: thirdElement].
      (choiceCollection thirdAt: firstIndex)
         add: 0].
   self displayInterface: title labels: labelsArray

MultipleListMultipleSelection methodsFor: private

allAssigned
   "return true if all roles have been assigned"

   globalList third do: [:element | (element at: 2) notNil
ifTrue: [(element at: 4) isZero ifTrue: [^false]]].
   globalList second do: [:tripleArray | tripleArray notNil
ifTrue: [tripleArray second do: [:array | (array at: 2)
second isEmpty ifTrue: [^false]]]].
   ^true

buildResponse
   "return a structure containing informations about
assignation.
   This structure is a TripleArray. Each element has form:
   task id         responsible id         (B)
   B) is a Double Array. Each element has the form:
   role id         (C)   {roles assigned to the task}
   C) ia an Array. Each element is a
   person id {assigned for the role}"

   | reply |
   reply := TripleArray new.
   1 to: globalList size do:
      [:index |
```

```
      | roles assignedRoles elem resp |
      elem := globalList thirdAt: index.
      (elem at: 2) notNil
         ifTrue: [resp := (elem at: 3)
                     secondAt: (elem at: 4)]
         ifFalse: [resp := nil].
      roles := globalList secondAt: index.
      assignedRoles := DoubleArray new.
      1 to: roles size do: [:roleIndex | assignedRoles add:
(roles thirdAt: roleIndex)
            with: ((roles secondAt: roleIndex)
                  at: 2) second].
      reply
         add: (elem at: 1)
         with: resp
         and: assignedRoles].
   ^reply

MultipleListMultipleSelection methodsFor: handling

endPressed
   "action taken because the end button is pressed"

   self allAssigned
      ifTrue:
         [[connectedTask selectionIs: self buildResponse]
fork.
         selectionWindow controller close]
      ifFalse: [WarningWindow text: 'Select one item in the
third lists for each item in the second lists']

firstList
   "gives to the interface the first list"

   ^globalList first

firstListSelectionIs: index
```

```
   "record first list selection"

   firstListSelection := index.
   self changed: #secondList

initialItem2A
   "return the initial item selection"

   firstListSelection isZero ifFalse: [^secondListASelection
at: firstListSelection].
   ^0

initialItem2B
   "return the initial item selection"

   firstListSelection isZero ifFalse: [^1].
   ^0

initialItem3C
   "return the initial item selection"

   firstListSelection isZero ifFalse: [^(globalList thirdAt:
firstListSelection)
         at: 4].
   ^0

secondListA
   "gives to the interface the list of items of selected
first list element"

   firstListSelection isZero ifFalse: [(globalList secondAt:
firstListSelection) notNil ifTrue: [^(globalList secondAt:
firstListSelection) first]].
   ^nil

secondListASelectionIs: index
   "store selected item"
```

```
   secondListASelection at: firstListSelection put: index.
   self changed: #thirdListAB

secondListB
   "gives to the interface the list of items of selected
first list element"

   firstListSelection isZero ifFalse: [((globalList thirdAt:
firstListSelection)
         at: 2) notNil ifTrue: [^Array with: ((globalList
thirdAt: firstListSelection)
               at: 2)]].
   ^nil

secondListBSelectionIs: index
   "store selected item"

   self changed: #thirdListC

thirdListA
   "gives to the interface the list of items of selected
first list element"

   firstListSelection isZero ifFalse:
      [| tmp |
      (tmp := secondListASelection at: firstListSelection)
isZero ifFalse: [^(((globalList secondAt:
firstListSelection)
            secondAt: tmp)
            at: 1) first]].
   ^nil

thirdListASelectionIs: index
   "move selected item from available persons' list to
selected persons' list"
```

```
   firstListSelection isZero | (secondListASelection at:
firstListSelection) isZero | index isZero
      ifFalse:
         [| tmp first second temp |
         tmp := secondListASelection at: firstListSelection.
         temp := (globalList secondAt: firstListSelection)
                  secondAt: tmp.
         first := (temp at: 1)
                  firstAt: index.
         second := (temp at: 1)
                  secondAt: index.
         (temp at: 1)
            removePosition: index.
         (temp at: 2)
            add: first with: second.
         self changed: #thirdListAB]

thirdListB
   "gives to the interface the list of items of selected
first list element"

   firstListSelection isZero ifFalse:
      [| tmp |
      (tmp := secondListASelection at: firstListSelection)
isZero ifFalse: [^(((globalList secondAt:
firstListSelection)
            secondAt: tmp)
            at: 2) first]].
   ^nil

thirdListBSelectionIs: index
   "move selected item from available persons' list to
selected persons' list"

   firstListSelection isZero | (secondListASelection at:
firstListSelection) isZero | index isZero
      ifFalse:
```

```
        [| tmp first second temp |
        tmp := secondListASelection at: firstListSelection.
        temp := (globalList secondAt: firstListSelection)
                 secondAt: tmp.
        first := (temp at: 2)
                 firstAt: index.
        second := (temp at: 2)
                 secondAt: index.
        (temp at: 2)
           removePosition: index.
        (temp at: 1)
           add: first with: second.
        self changed: #thirdListAB]

thirdListC
   "gives to the interface the list of items of selected
first list element"

   firstListSelection isZero ifFalse: [((globalList thirdAt:
firstListSelection)
         at: 3) notNil ifTrue: [^((globalList thirdAt:
firstListSelection)
            at: 3) first]].
   ^nil

thirdListCSelectionIs: index
   "store selected item"

   firstListSelection isZero ifFalse: [(globalList thirdAt:
firstListSelection)
         at: 4 put: index]


MultipleListMultipleSelection class
   instanceVariableNames: ''
```

```
MultipleListMultipleSelection class methodsFor: instance
creation

on: list for: aTask windowTitle: title labels: labelsArray
   "creates a new instance of MultipleListMultipleSelection
interface for the
   task aTask"

   ^super new
      selectFrom: list
      for: aTask
      windowTitle: title
      labels: labelsArray
```

```
DoubleListSelection subclass: #TripleListMultipleSelection
   instanceVariableNames: 'secondListSelection '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'User interfaces'



TripleListMultipleSelection methodsFor: initialization

selectFrom: choiceCollection for: aTask windowTitle: title
firstListLabel: label1 secondListLabel: label2
thirdListLabel: label3 selectedListLabel: label4
   "initialize the instance of the interface"

   connectedTask := aTask.
   globalList := choiceCollection.
   secondListSelection := Dictionary new.
   firstListSelection := 0.
   1 to: choiceCollection size do:
      [:firstIndex |
      | tripleArray2 |
      secondListSelection at: firstIndex put: 0.
      tripleArray2 := choiceCollection secondAt: firstIndex.
      1 to: tripleArray2 size do:
         [:secondIndex |
         | tmp doubleArray firstElement thirdElement |
         tmp := TripleArray new.
         firstElement := tripleArray2 firstAt: secondIndex.
         doubleArray := tripleArray2 secondAt: secondIndex.
         thirdElement := tripleArray2 thirdAt: secondIndex.
         1 to: doubleArray size do: [:thirdIndex | tmp
               add: (doubleArray firstAt: thirdIndex)
               with: false
               and: (doubleArray secondAt: thirdIndex)].
         tripleArray2
```

```
            at: secondIndex
            put: firstElement
            with: tmp
            and: thirdElement]].
   self
      displayInterface: title
      firstListLabel: label1
      secondListLabel: label2
      thirdListLabel: label3
      selectedListLabel: label4

TripleListMultipleSelection methodsFor: handling

displayInterface: title firstListLabel: label1
secondListLabel: label2 thirdListLabel: label3
selectedListLabel: label4
   "create the interface for the choice"

   | listView container endButton cancelButton |
   container := CompositePart new.
   selectionWindow := ScheduledWindow new.
   selectionWindow label: title.   "first selection list"
   listView := SelectionInListView
            on: self
            printItems: false
            oneItem: false
            aspect: #firstList
            change: #firstListSelectionIs:
            list: #firstList
            menu: nil
            initialSelection: nil
            useIndex: true.
   container add: (LookPreferences edgeDecorator on:
listView) noMenuBar borderedIn: ((LayoutFrame new)
leftFraction: 0; rightFraction: 0.3; topFraction: 0 offset:
20; bottomFraction: 1 offset: -30).   "first list title"
   container add: label1 asComposedText borderedIn:
```

```
((LayoutFrame new) leftFraction: 0; rightFraction: 0.3;
topFraction: 0; bottomFraction: 0 offset: 20).   "second
selection list"
   listView := SelectionInListView
            on: self
            printItems: false
            oneItem: false
            aspect: #secondList
            change: #secondListSelectionIs:
            list: #secondList
            menu: nil
            initialSelection: #initialItem
            useIndex: true.
   container add: (LookPreferences edgeDecorator on:
listView) noMenuBar borderedIn: ((LayoutFrame new)
leftFraction: 0.3; rightFraction: 0.6; topFraction: 0
offset: 20; bottomFraction: 1 offset: -30).   "second list
title"
   container add: label2 asComposedText borderedIn:
((LayoutFrame new) leftFraction: 0.3; rightFraction: 0.6;
topFraction: 0; bottomFraction: 0 offset: 20).   "third
selection list"
   listView := SelectionInListView
            on: self
            printItems: false
            oneItem: false
            aspect: #thirdList
            change: #thirdListSelectionIs:
            list: #thirdList
            menu: nil
            initialSelection: nil
            useIndex: true.
   container add: (LookPreferences edgeDecorator on:
listView) noMenuBar borderedIn: ((LayoutFrame new)
leftFraction: 0.6; rightFraction: 1; topFraction: 0 offset:
20; bottomFraction: 0.5).   "third list title"
   container add: label3 asComposedText borderedIn:
```

```
((LayoutFrame new) leftFraction: 0.6; rightFraction: 1;
topFraction: 0; bottomFraction: 0 offset: 20).   "fourth
selection list"
   listView := SelectionInListView
            on: self
            printItems: false
            oneItem: false
            aspect: #selectionList
            change: #selectionListSelectionIs:
            list: #selectionList
            menu: nil
            initialSelection: nil
            useIndex: true.
   container add: (LookPreferences edgeDecorator on:
listView) noMenuBar borderedIn: ((LayoutFrame new)
leftFraction: 0.6; rightFraction: 1; topFraction: 0.5
offset: 20; bottomFraction: 1 offset: -30).   "third list
title"
   container add: label4 asComposedText borderedIn:
((LayoutFrame new) leftFraction: 0.6; rightFraction: 1;
topFraction: 0.5; bottomFraction: 0.5 offset: 20).   "end
button"
   endButton := LabeledBooleanView new.
   endButton beTrigger.
   endButton controller beTriggerOnUp.
   endButton label: 'End'.
   endButton model: ((PluggableAdaptor on: self)
         getBlock: [:model | true]
         putBlock: [:model :value | model endPressed]
         updateBlock: [:model :value :parameter | true]).
   container add: endButton borderedIn: ((LayoutFrame new)
leftFraction: 0.25 offset: -25; topFraction: 1 offset: -25;
rightFraction: 0.25 offset: 25; bottomFraction: 1 offset:
-5).   "cancel button"
   cancelButton := LabeledBooleanView new.
   cancelButton beTrigger.
   cancelButton controller beTriggerOnUp.
```

```
   cancelButton label: 'Cancel'.
   cancelButton model: ((PluggableAdaptor on: self)
         getBlock: [:model | true]
         putBlock: [:model :value | model cancelPressed]
         updateBlock: [:model :value :parameter | true]).
   container add: cancelButton borderedIn: ((LayoutFrame
new) leftFraction: 0.75 offset: -25; topFraction: 1 offset:
-25; rightFraction: 0.75 offset: 25; bottomFraction: 1
offset: -5).   "displays the window"
   selectionWindow component: container.
   selectionWindow openNoTerminateIn: (150 @ 150 extent: 400
@ 200)

secondList
   "gives to the TripleListSelection interface the list of
items of selected first list
   element"

   firstListSelection isZero ifFalse: [^(globalList
secondAt: firstListSelection) first].
   ^nil

secondListSelectionIs: index
   "store selected item"

   secondListSelection at: firstListSelection put: index.
   self changed: #thirdList

selectionList
   "gives to the TripleListSelection interface the selection
list of items of
   selected second list element"

   firstListSelection isZero | (secondListSelection at:
firstListSelection) isZero
      ifFalse:
         [| tmp returnedList |
```

```
         returnedList := DoubleArray new.
         tmp := (globalList secondAt: firstListSelection)
                  secondAt: (secondListSelection at:
firstListSelection).
         1 to: tmp size do: [:index | (tmp secondAt: index)
               ifTrue: [returnedList add: (tmp firstAt:
index)
                     with: nil]].
         ^returnedList first].
   ^nil

selectionListSelectionIs: index
   "store selected item"

   index isZero
      ifFalse:
         [| tmp firstElement thirdElement |
         tmp := (globalList secondAt: firstListSelection)
                  secondAt: (secondListSelection at:
firstListSelection).
         firstElement := tmp firstAt: index.
         thirdElement := tmp thirdAt: index.
         tmp
            at: index
            put: firstElement
            with: false
            and: thirdElement].
   self changed: #thirdList.
   self changed: #selectionList

thirdList
   "gives to the TripleListSelection interface the list of
items of selected first list
   element"

   firstListSelection isZero | (secondListSelection at:
firstListSelection) isZero
```

```
      ifFalse:
         [| tmp returnedList |
         returnedList := DoubleArray new.
         tmp := (globalList secondAt: firstListSelection)
                  secondAt: (secondListSelection at:
firstListSelection).
         1 to: tmp size do: [:index | (tmp secondAt: index)
               ifFalse: [returnedList add: (tmp firstAt:
index)
                     with: nil]].
         ^returnedList first].
   ^nil

thirdListSelectionIs: index
   "store selected item"

   index isZero
      ifFalse:
         [| tmp firstElement thirdElement |
         tmp := (globalList secondAt: firstListSelection)
                  secondAt: (secondListSelection at:
firstListSelection).
         firstElement := tmp firstAt: index.
         thirdElement := tmp thirdAt: index.
         tmp
            at: index
            put: firstElement
            with: true
            and: thirdElement].
   self changed: #thirdList.
   self changed: #selectionList


TripleListMultipleSelection class
   instanceVariableNames: ''
```

```
TripleListMultipleSelection class methodsFor: instance
creation

on: list for: aTask windowTitle: title firstListLabel:
label1 secondListLabel: label2 thirdListLabel: label3
selectedListLabel: label4
   "creates a new instance of TripleListMultipleSelection
interface for the task
   aTask"

   ^super new
      selectFrom: list
      for: aTask
      windowTitle: title
      firstListLabel: label1
      secondListLabel: label2
      thirdListLabel: label3
      selectedListLabel: label4
```

```
ScheduledWindow subclass: #NotFailableTermination
   instanceVariableNames: 'connectedTask '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'User interfaces'



NotFailableTermination methodsFor: initialization

displayInterface
   "build and display interface"

   | container |
   container := CompositePart new.
   (Button
      label: ' Terminate '
      model: self
      execute: #terminated
      addTo: container)
      centerX: 0.25 centerY: 0.5.
   (Button
      label: '  Suspend  '
      model: self
      execute: #suspended
      addTo: container)
      centerX: 0.75 centerY: 0.5.
   self component: container.
   self openNoTerminateIn: (Window currentOrigin + 50
extent: 230 @ 60)

initialize: title for: task
   "initialize variables"

   self label: title.
   connectedTask := task.
```

```
   self controller: NoCloseController new.
   self displayInterface

NotFailableTermination methodsFor: handling

suspended
   "communicate user decision to the task"

   [connectedTask suspend] fork.
   self controller close

terminated
   "communicate user decision to the task"

   [connectedTask terminate] fork.
   self controller close


NotFailableTermination class
   instanceVariableNames: ''



NotFailableTermination class methodsFor: instance creation

of: title inTask: relatedTask
   "creates a new instance of the interface"

   ^super new initialize: title for: relatedTask
```

```
NotFailableTermination subclass: #FailableTermination
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'User interfaces'



FailableTermination methodsFor: initialization

displayInterface
   "build and display interface"

   | container |
   container := CompositePart new.
   (Button
      label: ' Terminate '
      model: self
      execute: #terminated
      addTo: container)
      centerX: 0.2 centerY: 0.5.
   (Button
      label: '  Suspend  '
      model: self
      execute: #suspended
      addTo: container)
      centerX: 0.5 centerY: 0.5.
   (Button
      label: '       Fail      '
      model: self
      execute: #failed
      addTo: container)
      centerX: 0.8 centerY: 0.5.
   self component: container.
   self openNoTerminateIn: (Window currentOrigin + 50
extent: 340 @ 60)
```

```
FailableTermination methodsFor: handling

failed
   "communicate user decision to the task"

   [connectedTask fail] fork.
   self controller close
```

Appendice D

Object Oriented Software Process Modeling

Mario Baldi, Silvano Gai, Maria Letizia Jaccheri
Dipartimento di Automatica e Informatica
Politecnico di Torino
10129 Torino
Italy

Software process models are complex artifacts produced by an engineering process, called PM meta-process. As the software production process, a PM meta-process consists of a set of phases such as analysis, design, implementation, operation, and maintenance. Here, an example process model is designed using the Coad/Yourdon methodology and implemented in SmallTalk.
Keywords: Process Modeling, Object Oriententation, Coad/Yourdon, Smalltalk.

D.1 Introduction

Software *Process Modeling* (PM) is the technique used to define, analyze, and automate software development activities [DNR91] [CFFS92] [Mad91]. A *Software Process* is the total set of software engineering activities needed to transform user requirements into operative software, and to evolve it. Software processes are typically life-cycle *activities* such as requirement analysis, design, coding, testing, installation, maintenance, etc. A *Process Model* provides a description of a class of software processes.

Software process models (PM [1]) are complex artifacts produced by an engineering process, called PM meta-process. A software meta-model describes a class of software meta-processes. A typical software meta-model is presented in figure D.1 and it consists of four phases that are `Analysis/Design`, `Implementation`, `Instantiation/Enactment`, and `Maintenance`.

In this work, an object oriented approach to process model design and implementation is investigated. An informal specification of an example process model is presented. From this specification a design is produced using a tool, called DECdesign [D.E92b], that implements the Coad/Yourdon [CY91a] [CY91b] approach to object oriented design. The design is then implemented in Smalltalk.

The existing PM systems may be classified [LC91] [ABGM92] according to three different paradigms: logic based, Petri nets, and process programming. According to the *process programming* paradigm, software process is defined by a program in a process programming language. Software development is thus the execution of this program [TBC+88]. Among the systems that adhere to the Process programming paradigm, the most interesting for us and here analyzed are those that exploit Object Oriented (OO) techniques for software process modeling [Sno92] [JLC92].

In the E^3 project an object oriented approach to process modeling is under investigation. The main advantages come from the fact that OO programming comes together with OO analysis and OO design techniques. Thus being a software process at least as complicate as a complex software system, OO techniques can be successfully applied to software process design and implementation.

The structure of this work is as follows. Section D.2 first introduces the

[1] In the following, when not ambiguous, PM will denote both process model and process modeling, i.e. the discipline of describing process models.

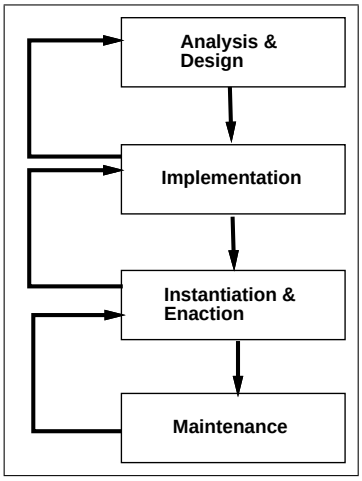

Figura D.1. A software meta-model

scenario problem specification, then it explains its design, and implementation. Hints on execution and maintenance are also given. Section D.3 illustrates the encountered problems and gives hints on their solution. Some conclusions are given in section D.4. Appendix D.5 summarizes the Coad/Yourdon notation, then appendix D.6 describes the main features of the design tool DECdesign, and appendix D.7 introduces the research project E^3 in which this work takes place.

D.2 The solution to a scenario problem

D.2.1 The Problem Description

The problem consists in designing, implementing, enacting, and changing a process model whose specification is:

```
Given a programmers team, there is one project manager.
Among the others, one third has to do design,
one third has to do review, and
one third has to program.
The design tool has to be chosen by the
project manager at enactment time.
The editor has to be chosen by its user.
The review, and programming phases may start in parallel,
but the product cannot be delivered unless
the review phase has been terminated.
```

The `Implementation` phase transforms the design specification given in a design specification language into an enactable one given in an executable PM language. Both the design specification language and the executable PM language depend from the chosen paradigm. The process performers are in charge of the `Instantiation/Enactment` phase, one of them playing the project manager role, some of them playing the designer, the reviewer and the programmer role.

The design tool is chosen by the project manager to be DECdesign.

The `Maintenance` phase consists of the re-engineering of the process description in order to upgrade it to new requirements. I.e. the `review` and `program` phases have to be sequentialized. This means that `program` cannot start before `review` has completed. In the case that `program` has already started before review is completed, `program` has to be re-initialized.

D.2.2 Design

The design of the problem solution consists of two main parts. The first is a set of general classes independent from this specific scenario problem. The other one defines the problem specific classes. The general classes, also called system defined classes, specify four PM sub-models: class `Task` is the root of the activity submodel, class `Role` is the root of the role submodel, class `Data` is the root of the product submodel, and class `Tool` is the root of the tool submodel. Figure D.2 shows a set of predefined classes and `instance connections`[2].

The Task submodel

A `Task` can be decomposed in subtasks. Subtasks may be either parallel or sequential; this is specified by the connection `successor` that define a partial order among tasks. The connection `father_child` links a father task to its children tasks while `feedback` links a (source) task to those tasks that have to be restarted due to the failure of the (source) task [3].

Task life passes through a number of states identified by the value of the attribute `status`. When a `Task` instance is created it enters the state `executing` and performs its activities that may consist of user interaction and

[2]In the following instance connection and relation are used as synonymous.

[3]All the above described task-to-task connections must be directed in order to distinguish the different roles played by the two objects involved.

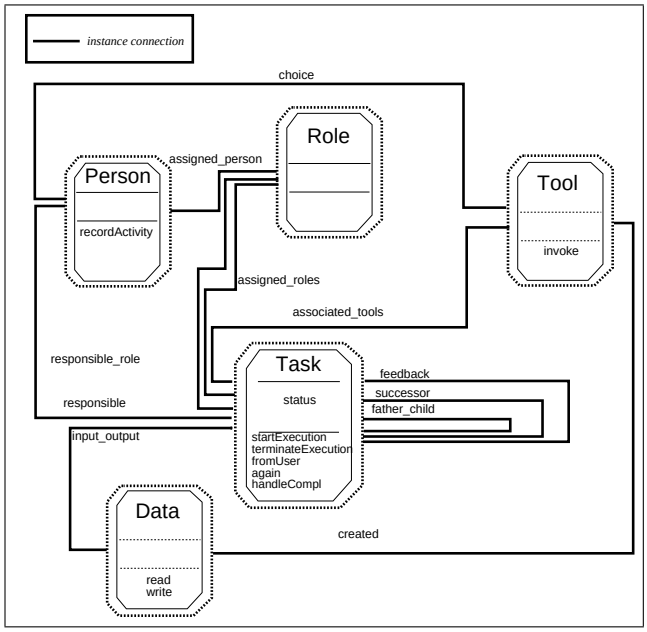

Figura D.2. The most general classes

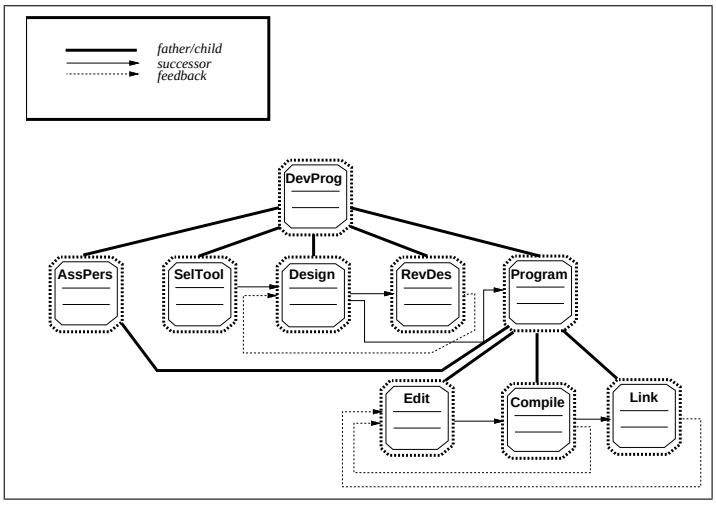

Figura D.3. The Task submodel

children instantiation. Children instantiation works as follows: a father task instantiates all the tasks that both are connected by `father_child` relations and whose predecessors (if any) are terminated. Task execution can either

terminate with failure or with success. In case of failure, the task sends the message `again` to all tasks related by `feedback` connection and enters the state `waitingPredecessors`. In case of success, if it is a composite task, it enters the state `handlingChildren`, else it enters the state `terminated`. When a child terminates, it notifies both its father, that provides for the instantiation of all successor tasks that may be instantiated, and it successors. A task in state `waitingPredecessors` executes when all its predecessors have terminated. A composite task in state `handlingChildren` terminates when all its children terminate.

Figure D.3 shows the `Task` subclasses used to design the scenario problem: `DevProg` is connected by `father_child` [4] relation to `AssPers`, `SelTool`, `Design`, `RevDes`, and `Program`. `Program` is again linked to `Edit`, `Compile`, and `Link`.

The Role submodel

Figure D.4 shows the `Role` subclasses needed for the scenario problem solution. Class `Role` gives information needed to bind a task instance to the `Person` instance responsible for the task (relation `responsible`).

Class `Role` is not characterized by the services it offers, but by the relations it participates to. A `Task` instance is connected by relation `assigned_roles` to all those roles that will be responsible for its child tasks. A `Role` instance is connected by relation `assigned_person` to the `Person` instances that play the given role.

A `responsible_role` relation connecting a `Task` subclass `T1` to a `Role` subclass `R1` imposes the constraint that `T1` responsible must play the `R1` role. In figure D.4 under each `Role` subclass are listed the connected `Task` subclasses.

When a task decomposes itself into subtasks it must connect them to their responsible persons using the connection `responsible`. The father task selects a set of candidate responsibles among the persons assigned to it. Then, it creates an instance of the system defined class `Ass_Tasks` that interacts with the user to choose a responsible (e.g., see figure D.5).

The Product submodel

[4] This is an instance connection that is depicted in a non standard way for the sake of understandability.

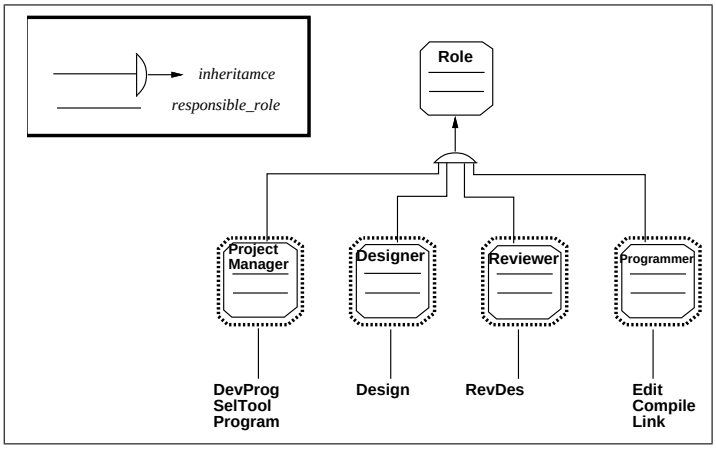

Figura D.4. The Role submodel

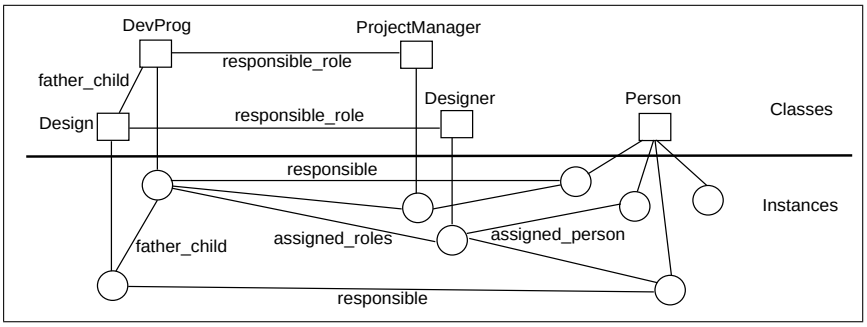

Figura D.5. An example of role usage

Tasks might take as input and produce as output instances of class `Data` (or of one of its subclasses); `Data` objects are connected to task instances by the connection `input_output`.

Figure D.6 shows the product model: class `Config` models product configurations whose components are instances of class `SWObj` that represents software objects like files. The relation `component` links a configuration with its components.

`ReqDoc` models requirement documents whereas `DesDoc` models design documents. The relation `describe` connects the requirement documents to the respective design documents.

Class `Compil` is subclass of `SrcFile` and it represents source files that

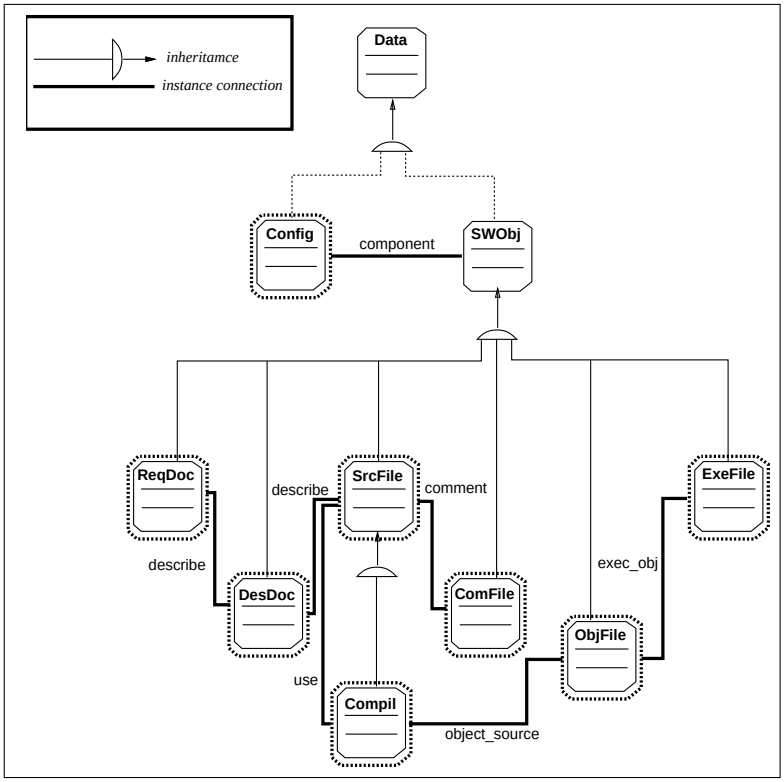

Figura D.6. The Data submodel

may be compiled. A compilable file may include some source files that are connected by relation `use`. A source file is linked by the `describe` connection to the respective design document. A source file may be associated to a comment file represented by an instance of `ComFile` class connected by relation `comment`.

Class `ObjFile` models object files that are obtained compiling source files connected by relation `object_source`. Compilation involves objects connected by relation `use`. Instances of `ExeFile` class represent executable files that are generated linking together the object files connected by `exec_obj`.

The Tool submodel

`Tool` subclasses needed for the problem solution are shown in figure D.7. Instance connection `associated_tools` (fig. D.2) denotes the tools that are available to a task for producing its output. When a task generates subtasks, they inherit the tools association of their father. If, for a given task, a tool

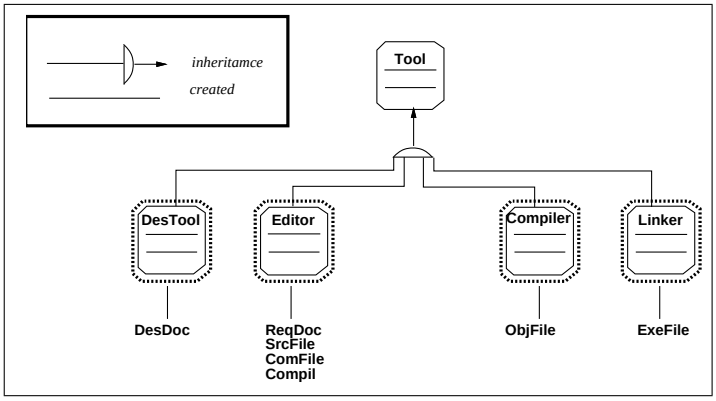

Figura D.7. The Tool submodel

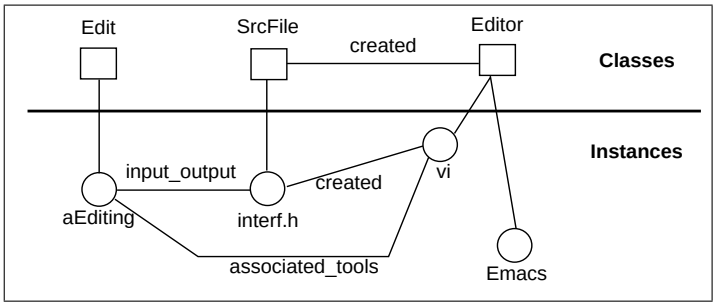

Figura D.8. The use of `created` connection

has to be specified at enactment time, the system defined task `SelTool` has to be instantiated before the given task. `SelTool` interacts with the user to choose one tool for each category. This tool will be the only one connected by the relation `associated_tools`.

Relation `created` relates `Tool` subclasses to `SWObj` subclasses defining which kind of tool can manipulate a software object. In figure D.7 under each `Tool` subclass are listed related `SWObj` subclasses. A `SWObj` instance is connected to the `Tool` instance that has created it by relation `created` (e.g. see figure D.8).

If a task needs a tool to produce its output, it finds it by querying the connection `created` of its output `SWObj`. If more than one tool of the specified kind is associated to a task by connection `associated_tools`, the responsible has to choose one of them.

About relations

Relations that have been introduced before, fall in three conceptual categories:

1. *instance_to_instance*: they interconnect instances and they have been modeled by Coad/Yourdon instance connections. E.g., relation `responsible` connects `Role` objects to `Task` objects.

2. *class_to_class*: they interconnect classes. The *inheritance* relation has been directly modeled by the Coad/Yourdon *gen/spec* structure connection. `Responsible_role` is another class_to_class relation.

3. *class_to_class* and *instance_to_instance*: they interconnect both classes and objects. Interconnections between classes constraint interconnections between objects. E.g., `father_child` relation connects both a class `Task` to the classes of its children and a `Task` object to its children objects.

The last two kinds of relations have been modeled by instance connections as the Coad/Yourdon notation does not allow the user to define connections between classes. Though, in the SmallTalk implementation this kind of diversification has been explicitly implemented as it will be shown in section D.2.3.

D.2.3 Implementation

The above described PM has been implemented using Objectworks(r)/ Smalltalk [Sys90], Release 4 under MS-Windows 3.0. In this section, the implementation of relations is given. Then task implementation and finally hints of user interface implementation are given too.

Relations

The relation concept has been implemented by attributes representing identifiers of the related objects [R$^+$91].

1. *instance_to_instance* connections are implemented using attributes that contain reference to the connected objects.

2. *class_to_class* connections are implemented using class attributes. Due to the Smalltalk inheritance mechanism, it is necessary to initialize these class attributes as `Dictionary` instances in the highest superclass. Then, a new entry is created for each subclass that has a connection.

3. *class_to_class* and *instance_to_instance* are implemented by a combination of the above two mechanisms. The semantics of the constraints imposed by class_to_class relations on instance_to_instance relations have been expressed and implemented in the methods that create connections between objects.

Task class

```
Object subclass: #Task
    instanceVariableNames:  'status associatedTools father children
        assignedRoles responsible successor predecessors waitingFeedback
        input output userAction'
    classVariableNames:  'Children Successors Predecessors
        WaitingFeedback Input Output ResponsibleRole
    poolDictionaries:  ''
    category:  'Tasks'
Task methodsFor:  'initialization'
...
    initialize: father
Task methodsFor:  'standard'
    again
    changeStatus: newStatus
    fromUser: matter
    handleCompletion: aTask
    startExecution
    terminateExecution
Task methodsFor:  'relation handling'
...
Task methodsFor:  'private'
...
Task methodsFor:  'communication'
...
Task methodsFor:  'children instantiation'
    instantiateChildren
    instantiate: number childrenOfClass: childClass
    isPossibleToInstantiate: number childrenOfClass: childClass
...
```

Figura D.9. The `Task` class.

A simplified definition of class Task is given in figure D.9. The attribute `status` stores the current task execution status. The attribute `userAction` is a semaphore to synchronize task execution and user interaction.

Class variables are used to implement relations. `Children` class variable implements the relation `father_child`. This is a data dictionary containing one entry for each child class. Each entry contains both the identifier of the child class, and an integer denoting the number of instances to be instantiated. If this number is not known a priori (it may depend upon the execution of the father task or upon the product structure), zero is stored in the dictionary entry.

```
instantiateChildren
        "instantiates children if possible"
    Children childrenAndNumberOf:  self class do:  [:childClass :num | (num isZero
        ifTrue:  [self isPossibleToInstantiateChildrenOfClass:  childClass]
        ifFalse:  [self isPossibleToInstantiate:  num childrenOfClass:  childClass])
        ifTrue:  [num isZero
            ifTrue:  [self instantiateChildrenOfClass:  childClass]
            ifFalse:  [self instantiate:  num childrenOfClass:  childClass]]]
```

Figura D.10. The `Task` class **instantiateChildren** method

In the following the most significative methods are described:

- `instantiateChildren` (fig.D.10) is a `Task` private method that is defined in class `Task` and is not redefined in subclasses thanks to its parametricity obtained using *class_to_class* connections.

- `isPossibleToInstantiate:childrenOfClass` implements *incremental instantiation*. Subtasks are not instantiated until they are not ready to be executed, e.g. all predecessors are in `completed` state. This method takes as parameter the class identifier of the task to be instantiated and finds out the predecessors classes by querying class relations. It then identifies instances of those classes that really have to be related as predecessors of the new instance. These are selected in a different way depending on whether the number of instances is statically specified or not.

- `instantiate:childrenOfClass:` (fig. D.11) instantiates a fixed number of children of a specified class. This method first instantiates new children tasks. Each child is connected by `father_child` instance connection to its father then it is connected to both its input and output data invoking the private method `connectIOOf:`. The same tools associated to the father task are linked to the child task by the message

```
instantiate: number childrenOfClass: childrenClass
        "instantiates number children of the class childrenClass"
    number
        timesRepeat:
            [| newChild |
            children add:  (newChild := childrenClass new:  self).
            self connectIOOf:  newChild.
            newChild associatedTools:  associatedTools.
            self connectSiblingsOf:  newChild.
            self assignFor:  newChild]
```

Figura D.11. The `Task` class **instantiate:childrenOfClass:** method

`associatedTools:` sent to the new child. Then the instantiated task is connected to its predecessors and to those tasks (if any) that eventually wait feedback from it. Finally the private method `assignFor:` is invoked to connect the child to both its responsible and assigned persons.

```
startExecution
        "begin review design by notifying the responsible"
    super startExecution.
    responsible
            recordActivity:  'Review product design'
            from:  self
            matter:  #startActivity
            description:  'Review design of product in development'
```

Figura D.12. The `RevDes` class **startExecution** method

- method `startExecution` is implemented in class `Task` and redefined in each subclass. Class `RevDes` implementation of this method is shown in figure D.12.

```
fromUser: matter
        "gets attention from responsible for matter"
    super fromUser:  matter.
    matter = #startActivity
        ifTrue:
            [self runTools.
            self openTerminationInterface].
    matter = #suspend ifTrue:  [responsible
            recordActivity:  'Review product design'
            from:  self
            matter:  #resume
            description:  'Continue reviewing design of product in development'].
    matter = #resume
        ifTrue:
            [self runTools.
            self openTerminationInterface].
    matter = #fail
        ifTrue:
            [self giveFeedback]
```

Figura D.13. The `RevDes` class **fromUser:** method

- method `fromUser` (fig. D.13) is defined in class `Task` to specify standard action to be taken for execution termination. It is redefined in each subclass to specify actions typical of specific task execution. Method

```
Model subclass: #Agenda
    instanceVariableNames:  'selection list listSemaphore relatedPerson '
    classVariableNames:  ''
    poolDictionaries:  ''
    category:  'User interfaces'
Agenda methodsFor:  'initialization'
    assignTo: aPerson
...
Agenda methodsFor:  'list menu'
...
Agenda methodsFor:  'comunication'
    record: aLabel from: aTask matter: aSymbol description: aText
Agenda methodsFor:  'handling'
...
```

Figura D.14. The `Agenda` class.

```
assignTo: aPerson
        "initialize the Agenda"
    list := DoubleArray new.
    listSemaphore := Semaphore forMutualExclusion.
    relatedPerson := aPerson.
    selection := 0.
    self displayWithTitle:  'Agenda of ' , aPerson provideName
```

Figura D.15. The `Agenda` class **assignTo:named:** method

`fromUser` is called by the user interface and receives as parameter the reason for which the UI has been invoked. According to this parameter, `fromUser` takes a different action.

The Agenda

Each PM performer communicates with the enacting PM using user interfaces. The main UI is called `Agenda` and it is simply an activity list in which the user can select the next activity to perform.

Figure D.14 shows class `Agenda` definition. Its attributes include `list` (a data structure containing informations to manage the activity list), `selection` (an integer identifying selected item in the list), and `listSemaphore` (an instance of `Semaphore` used to protect `list` which is accessed both by the interface and by the task).

The main methods for class `Agenda` are the following:

- `assignTo:` (fig. D.15) initializes the attributes, connects the associated

user, and displays the interface itself.

```
record: elementName from: taskId matter: aMatter description: descr
        "records a new activity received from Task taskId for aMatter
        fromUser of taskId task is invoked giving back the symbol
        aMatter when the user selects this activity''
    | temp |
    temp := Array with:  descr with:  taskId with:  aMatter.
    listSemaphore critical:  [list add:  elementName with:  temp].
    self changed:  #agenda
```

Figura D.16. The `Agenda` class **record:from:matter:description:** method

- `record:from:matter:description:` (fig. D.16) appends a new item labeled `elementName` to the activity list and notifies the interface controller about the change. This is invoked when task execution requires responsible interaction. Parameter `descr` contains a short description of the activity. This text is shown to the user when he requires it (fig. D.17). When the user executes the activity in the `Agenda` list, the respective task is notified by method `fromUser`.

D.2.4 Instantiation and Enactment

To instantiate the PM is necessary to create the instances of the class `Person` that model the PM performers, the instances of the roles played by these

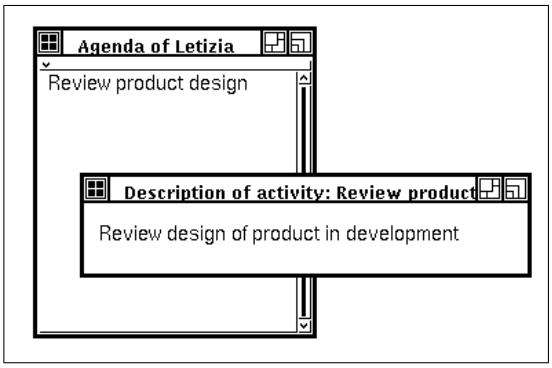

Figura D.17. The Agenda

persons, and the objects representing the available tools. Then, it is possible to start enactment creating the instance of the higher level task (`DevProg` in our case) and connecting it to all the above mentioned objects. Task initialization invokes the service `startExecution`. Method `startExecution` instantiates task children, if possible.

The enactment is based upon message passing among objects involved in the PM. There is a control thread for each interface running. In future expansions of this prototype, it will be possible to create more control threads, thus obtaining a deeper degree of concurrency to better simulate a real PM enactment environment.

D.2.5 Maintenance

An OO system with explicit class representation is a reflective [Mae88] system. This means that both the state and the description of the system can be inspected and manipulated. A PM maintenance tool has been designed and implemented by one of the author for the EPOS system [JC93]. This tool highly interacts with the user to help him in evaluating the impacts of a given change, to actually perform class changes, and to restore consistency after changes. The reflectiveness of Smalltalk makes it possible to implement the same kind of tool on top of this system. This is in fact one of our objective for the future.

D.3 Encountered problems and proposed solutions

To yield PM implementation more efficient it is useful to have class level services and connections. In an analogous way PM designs can be more effective if it is possible to model such items.

Thus in the design of the scenario problem solution we gave an extended semantics to the Coad/Yourdon notation. I. e. we supposed that Instance Connection does not simply model mapping between objects, but between their classes too, as explained in section D.2.3.

We also supposed that each instance connection is bidirectional. However, this is not always required and then it can be helpful to be able to specify the direction of a connection.

D.4 Conclusions

The solution of this example process model has resulted in a set of system defined classes and in a set of domain specific classes. The first set may be seen as a PM kernel for the specification of even more complex process models. On the other hand, even the second set may be seen as a PM reusable library. The next step is of course to design and implement a new PM and to assess it.

Then, to make PM construction more effective, a set of meta-tools that facilitate the process of building, reusing, and validating process fragments has to be build on top of the existing prototype.

Towards PM enaction, the prototype has to be extended with distribution, DBMS and tools integration, and project management facilities.

D.5 The Coad/Yourdon Notation

A Coad/Yourdon model is structured in *five layers* (Subject, Class&Objects, Structure, Attribute and Service) to apply the whole-part method of organization. Then, in an orthogonal way, it is divided in *four components* (Problem Domain, Human Interaction, Task Management and Data Management). Each component structure reflects the five layers and every layer provides a different view of the model. Only the Problem Domain Component is presented in the design of the scenario problem.

In accordance with the five layers, the Coad/Yourdon approach to Object Oriented analysis (OOA) and design (OOD) consists of five major *activities*. These are activities and not sequential steps and, the order among them is not strictly imposed.

Finding Class&Objects consists of identifying both the classes needed to model the problem and the objects belonging to them. The Coad/Yourdon notation provides the `Class` symbol representing a class that does not have any instance (sometimes referred to as an *abstract class*), which consists of a rounded bold rectangle divided into three horizontal sections (see figure D.4). Whereas the `Class&Objects` symbol consists of a rounded bold rectangle divided into three horizontal sections representing the class, and an external light rectangle representing its Objects (see figure D.2).

Identifying Structures leads to establish two kinds of relationships between classes. `Generalization-Specialization` structure represents the *subclass* (*inheritance*) concept; it is represented as a semicircle with a line drawn outward from its midpoint to the more general class and a line drawn to each of its specialization classes (see figure D.4). As `Generalization-Specialization` structure reflects a mapping between classes, the endpoints must be connected to the bold rectangle of the symbols.

The other kind of structure is the `Whole-Part` structure whose symbol consists of a triangle with a line drawn outward from the top to the whole object and a line drawn outward from the base to the part object. The endpoints of the lines are connected to the light rectangle of symbols in order to reflect a mapping between objects. Each end of a `Whole-Part` structure is marked with a cardinality specifying the number of parts that a whole has and the number of wholes a part may belong to.

Identifying Subjects permits to group the `Class` and `Class&Objects` symbols into `Subjects`. This helps in controlling visibility and guiding reader attention. A `Subject` can contain `Classes, Class&Objects` and other `Subjects` providing the possibility of defining different levels of abstraction.

The methodology provides three different notations for a subject:

- collapsed: a light rectangle enclosing the order number and the name of the `Subject`;
- partially expanded: a light rectangle enclosing the number and the name of the `Subject` and the list of the items that are included in it;
- expanded: a light rectangle, with the number of the `Subject`, drawn around the items included in the `Subject`.

Defining Attributes `Attributes` may be placed in the center section of the `Class&Objects` and `Class` symbols (see figure D.2). Here the endpoints of `Instance Connections` are also attached. `Instance Connections` provide the mappings between Objects to model *association*.

`Instance Connections` are drawn as lines connecting the light rectangles of two Objects (see figure D.2). Each end is marked with a cardinality that

represents the number of Objects that may be connected[5].

Defining Services `Services` are placed in the bottom section of the `Class&Objects` and `Class` symbols (see figure D.2) and represent the interface of the Object or the Class (to create new instances).

Objects interact by message passing. The mapping between the sender and the receiver of a message is modeled by a `Message Connection` drawn as an arrow from the sender to the receiver. The endpoints of the arrow are attached at the bottom section of the symbols.

Part of the service definition activity is the construction of `Object State Diagrams` which describe the different states of an Object over time. The notation of the state is a rectangle enclosing the characterization of the state. An arrow connecting two states represents a state transition.

Finally, `Services` may be specified using `Service Charts` that are a sort of flow chart. Symbols for `Condition, Text` and `Loop Block` are provided with a `Connector` symbol to be put between them.

As evident from this short description of the notation, this methodology provides means to represent the class level of the model, but not at all the instance level of it.

D.6 DECdesign

The design tool DECdesign V2.0-001 [D.E92b] has been used under the ULTRIX 4.2 operating system. DECdesign provides support for five design methodologies.

The notation used by DECdesign implementation of the Coad/Yourdon method is slightly different from the one described in [CY90].

`Instance Connections` endpoints do not connect to the light rectangle of `Class&Objects` symbols, but to the internal Class symbol.

On the contrary `Generalization-Specialization` structure endpoints connect to the rectangle representing objects of the class instead of connecting to the inner one.

Although we used DECdesign to design the solution of the scenario problem, diagrams presented in this paper have not been done using this tool and they follow the notation described in [CY90]. In fact DECdesign models

[5]In this document, cardinalities are omitted for space reasons.

print-out have fixed dimensions (very small or very big) and thus are not easy to include in a paper.

The tool was helpful particularly in both handling item descriptions and connecting objects at the proper points.

D.7 The E^3 Project

E^3 is a research project in the software process domain. The project was begun in July 1992 at Dipartimento di Automatica e Informatica of Politecnico di Torino, Italy and now involves eight persons: a professor, a researcher, a PhD student, two graduated research assistants and three ungraduated students.

The E^3project has two main aims: the first is process knowledge acquisition from the real world processes to produce significant process models, the second is to build a PM prototype to experiment PM architectural issues.

To model real world processes one needs methods for process analysis and design together with a PM system for simulating, enacting, and validating the established model. OO analysis/design methods, such as Booch and Coad/Yourdon, have been applied to the specification of the ISPW6 [KFF+90] example problem. During this phase, an automatic design tool, DECdesign has been used to automate the PM design process. The resulting process model has been prototyped and simulated in Smalltalk. Temporal logic has been exploited as well to provide a formal description of a scenario software process. The next step is to scale up to the modeling of semi-real software processes such as the one recommended by the ISO-9000 quality assurance standard.

The solution to the scenario problem presented in this paper has been developed by one ungraduate student under the supervision of a researcher.

The requirement phase of the E^3 PM system is now about to conclude. The requirements have mainly stemmed from studying the PM meta-process, i.e. the process of producing, instantiating, enacting, and observing process models. First, a PM system has to assist the software engineers in charge of analyzing and designing process models. Thus it has to provide one or a set of formalisms to significantly represent process model fragments and their inter-relationships. Third, a PM system has to provide assistance during the instantiation of a process model, as a static description, into an enacting

model. Then, it must assist and control the interaction of the process end-users, i.e. the software practitioners, with the enactable models. Last, but not least, it has to provide room for change.

Bibliografia

[ABGM92] P. Armenise, S. Bandinelli, C. Ghezzi, and A. Morzenti. Software Process Representation Languages: Survey and Assessment. In *Proc. 4th IEEE International Conference on Software Engineering and Knowledge Engineering, Capri, Italy, June 17-19. 31 pages*, June 1992.

[ACM90] V. Ambriola, P. Ciancarini, and C. Montangero. Software Process Enactment in Oikos. In *Proceedings of the 4th ACM SIGSOFT Symposium on Software Development Environments, Irvine, California*, pages 183–192, 1990.

[B+89] K. Benali et al. Presentation of the ALF project. In *[MSW90]*, page 23, May 1989.

[BEM91] N. Belkhatir, J. Estublier, and W. L. Melo. ADELE2 - An Approach to Software Development Coordination. In *[FCA91]*, pages 89–100, 1991.

[BFG91] Sergio Bandinelli, Alfonso Fuggetta, and Carlo Ghezzi. Software process as real-time systems: A case study using high level petri nets. In *[FCA91]*, 1991.

[BK92] Naser S. Barghouti and Gail E. Kaiser. Scaling up rule-based development environments. *International Journal on Software Engineering and Knowledge Engineering, World Scientific*, 2(1):59–78, March 1992.

[Boo91] Grady Booch. *Object Oriented Design with Application.* The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1991.

[C+89] Reidar Conradi et al., editors. *Norsk Informatikk Konferanse — NIK'89*, Trondheim, Norway, November 1989. Tapir. (Stavanger Airport Hotel, Sola, 14-15 Nov. 1989).

[CFFS92] Reidar Conradi, Christer Fernstrom, Alfonso Fuggetta, and Robert Snowdon. Towards a Reference Framework for Process Concepts. In *J.-C. Derniame (ed.): Proc. from EWSPT'92, Trondheim, Norway, Springer Verlag LNCS*, September 1992.

[Cur89] Bill Curtis. But You Have to Understand, This Isn't the Way We Develop Software at Our Company. Technical report, MCC, 1989. Technical Report Number STP-203-89.

[CWL+89] Reidar Conradi, Per H. Westby, Anund Lie, Ole Solberg, Vincenzo Ambriola, and M. Letizia Jaccheri. Software process management in EPOS. In *[C+ 89], p. 213–228*, 1989. (rev. July/Oct 1989. Also as DCST TR 26/89 — STF40-A89147 ISBN 82-595-5739-8 — EPOS TR 83; and presented at IFIP WG2.4 meeting, Warsaw, 18–22 Sept 1989).

[CY90] Peter Coad and Edward Yourdon. *Object-Oriented Analysis.* Prentice Hall, Englewood Cliffs, first edition, 1990.

[CY91a] Peter Coad and Edward Yourdon. *Object-Oriented Analysis.* Prentice Hall, Englewood Cliffs, second edition, 1991.

[CY91b] Peter Coad and Edward Yourdon. *Object-Oriented Design.* Prentice Hall, Englewood Cliffs, second edition, 1991.

[D.E92a] D.E.C. Dec aca services: System integrator and programmer guide, April 1992. Digital Equipment Corporation, Maynard, Massachusetts.

[D.E92b] D.E.C. Guide to decdesign, August 1992. Digital Equipment Corporation, Maynard, Massachusetts.

[DG90] Wolfgang Deiters and Volker Gruhn. Managing Software Processes in the Environment MELMAC. In *Proc. of the 4th ACM SIGSOFT Symposium on Software Development Environments, Irvine, California. In ACM SIGPLAN Notices, Dec. 1990*, pages 193–205, December 1990.

[DNR91] Mark Dowson, Brian Nejmeh, and William Riddle. Fundamental Software Process Concepts. In *[FCA91]*, pages 15–37, 1991.

[Dow87] Mark Dowson. ISTAR and the Contractual Approach. In *Proc. of the 9th Int'l ACM-SIGSOFT/IEEE-CS Conference on Software Engineering, Monterey, CA*, pages 287–288, April 1987.

[EJP+91] W. Emmerich, G. Junkermann, B. Peuschel, W. Schäfer, and S. Wolf. MERLIN: Knowledge-based Process Modeling. In *[FCA91]*, pages 181–187, 1991.

[FCA91] Alfonso Fuggetta, Reidar Conradi, and Vincenzo Ambriola, editors. *Proceedings of the First European Workshop on Process Modeling (EWPM'91)*, CEFRIEL, Milano, Italy, 30–31 May 1991, 1991. Italian Society of Computer Science (AICA) Press.

[GFM+91] Carlo Ghezzi, Alfonso Fuggetta, Sandro Morasca, Angelo Morzenti, and Mauro Pezzè. *Ingegneria del software: progettazione, sviluppo e verifica*. Mondadori Informatica S.p.A., Milano, Italy, 1991.

[GR83] Adele Goldberg and Dave Robson. *Smalltalk-80: The Language and its Implementation*. Addison Wesley, 1983.

[H+88] D. Harel et al. Statemate: A working environment for the development of complex reactive systems. In *[IEE88]*, pages 396–406, 1988.

[Hen88] Peter B. Henderson, editor. *Proc. of the 3rd ACM SIGSOFT/ SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Boston), *257 p.*, November 1988. In ACM SIGPLAN Notices 24(2), Feb. 1989.

[HK89] Watts S. Humphrey and Marc I. Kellner. Software process modeling: Principles of entity process models. In *Proc. of the 11th Int'l ACM-SIGSOFT/IEEE-CS Conference on Software Engineering, Pittsburgh, PA*, 1989.

[IEE88] IEEE/ACM, editor. *Proc. of the 10th Int'l ACM-SIGSOFT/ IEEE-CS Conference on Software Engineering*, Singapore, April 1988.

[Jac91] M. Letizia Jaccheri. Process Modeling: Concepts, Paradigms, and the EPOS Solution, October 1991. (Accepted at 5th Nordic Workshop on Programming Environment Research – NWPER'92, 8–10 Jan. 1992, Tampere, Finland).

[JC93] Maria Letizia Jaccheri and Reidar Conradi. Techniques for Process Model Evolution in EPOS. Technical report, Politecnico di Torino, DAI, January 1993.

[JLC92] Maria Letizia Jaccheri, Jens-Otto Larsen, and Reidar Conradi. Software Process Modeling and Evolution in EPOS. In *Proc. 4th IEEE International Conference on Software Engineering and Knowledge Engineering, Capri, Italy, June 17-19.*, 1992.

[Jr.90] Stanley M. Sutton Jr. A flexible consistency model for persistent data in software-process programming languages. In *Proc. of the 4th International Workshop on Persistent Object Systems*, pages 297–310, September 1990.

[Kat89] T. Katayama. A Hierarchical and Functional Software Process Description and its Enaction. In *Proc. of the 11th Int'l ACM-SIGSOFT/IEEE-CS Conference on Software Engineering, Pittsburgh, PA*, pages 343–352, 1989.

[Kat90] Takuya Katayama, editor. *Support for the Software Process, Proc. of the 6th International Software Process Workshop, 1990*, Hakodate, Japan, October 1990. IEEE Computer Society Press.

[KFF+90] Marc I. Kellner, Peter H. Feiler, Anthony Finkelstein, Takuya Katayama, Leon Osterweil, Maria Penedo, and H. Dieter Rombach. Software Process Modeling Problem (for ISPW6), August 1990.

[LC91] Chunnian Liu and Reidar Conradi. Process Modeling Paradigms: an Evaluation. In *[FCA91]*, pages 39–52, 1991.

[LC93] Chunnian Liu and Reidar Conradi. Automatic Replanning of Task Networks for Process Model Evolution in EPOS. In *Ian Sommerville (Ed.): "Proc. from the 4th European Software Engineering Conference (ESEC'93)", Garmisch-Partenkirchen, FRG. Forthcoming as a Springer LNCS. 17 p*, September 1993.

[LH89] Lung-Chun Liu and Ellis Horowi¡tz. A formal model for software project management. *IEEE Transactions on Software Engineering*, 15(10):1280–1293, October 1989.

[Lon92] Jacques Lonchamp. A Structured Conceptual and Terminological Framework for Software Process Engineering. Technical report, Centre de Recherche en Informatique de Nancy (CRIN), June 1992.

[Mad91] Nazim H. Madhavji. The process cycle. *Software Engineering Journal*, 6(5):234–242, September 1991.

[Mae88] P. Maes. *Issues in Computational Reflection*, pages 21–35. North Holland, 1988.

[MSW90] N. Madhavji, W. Schaefer, and H. Weber, editors. *Proc. of the First International Conference on System Development Environments and Factories — SDEF'89, 9-11 May 1989, Berlin*, London, March 1990. Pitman Publishing, 241 p.

[Ost87] Leon Osterweil. Software Processes are Software Too. In *Proc. of the 9th Int'l ACM-SIGSOFT/IEEE-CS Conference on Software Engineering, Monterey, CA*, pages 2–13, March 1987. (Keynote address at the conference).

[Per89] Dewayne E. Perry, editor. *Experience with Software Process Models, Proc. of the 5th International Software Process Workshop, 1989*, Kennebunkport, Maine, USA, October 1989. IEEE Computer Society Press.

[R+91] James Rumbaugh et al. *Object-Oriented Modeling and Design.* Prentice Hall, 500 p., 1991.

[Rob81] David Robson. Object Oriented Software Systems. *Byte*, 6(8):74, 1981.

[Rom90] H.Dieter Rombach. A framework for assessing process representations. In *[Kat90]*, pages 175–185, October 1990.

[Rom91] H.Dieter Rombach. A mvp-l: A language for process modeling in-the-large. submitted for publication in IEEE Transactions on Software Engineering, 1991.

[SB86] Mark Stefik and Daniel G. Bobrow. Object Oriented Programming: Themes and Variations. *AI Magazine*, 6(4):41, 1986.

[Sch90] Wilhelm Schaefer. Specimen presentation. Technical report, Eureka Software Factory (ESF) Conference, Berlin, November 1990.

[Sno92] Robert Snowdon. An Example of Process Change. In *J.-C. Derniame (ed.): Proc. from EWSPT'92, Sept. 7–8, Trondheim, Norway, Springer Verlag LNCS 635*, pages 178–195, September 1992.

[Ste87] Lynn Andrea Stein. Delegation is inheritance. In *Proc. of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'87)*, pages 138–146, Kissimmee, Florida, October 1987. In ACM SIGPLAN Notices 22(12), Dec. 1987.

[Sug90] Yasuhiro Sugiyama. *Object Process Modeling and the Process Programming Language Galois*. PhD thesis, Computer Science Department, University of Southern California, Los Angeles, CA, April 1990.

[Sys90] ParcPlace Systems. User's guide, 1990. ParcPlace Systems, Mountain View, California.

[TBC+88] Richard N. Taylor, Frank C. Belz, Lori A. Clarke, Leon Osterweil, Richard W. Selby, Jack C. Wileden, Alexander L. Wolf, and Michael Young. Foundations for the Arcadia Environment Architecture. In *[Hen88]*, pages 1–13, November 1988.

[War89] Brian Warboys. The IPSE 2.5 Project: Process Modelling as the basis for a Support Environment. In *[MSW90], 26 p.*, May 1989.

[YE89] Y.Sugiyamaa and E.Horowitz. Opm: An object process modeling environment. In *[Per89]*, October 1989.

[YE90] Y.Sugiyamaa and E.Horowitz. Language support for object process modeling. In *[Kat90]*, pages 195–198, October 1990.