Università degli Studi di Bologna

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI

Corso di Laurea in Informatica

PER UN NUOVO ORIENTAMENTO NELLA PROGETTAZIONE DEI LINGUAGGI DI PROGRAMMAZIONE

Tesi di laurea di:

RICCARDO SOLMI

Relatore:

Prof. ANDREA ASPERTI

Keywords: adaptable behavior; class refactoring; object oriented programming criticism; programming paradigms comparison; foundations of programming languages

Sessione II

Anno Accademico 1998 – '99

SOMMARIO

INDICE DELLE FIGURE E DELLE TABELLE

Capitolo 1

INTRODUZIONE

I linguaggi di programmazione nascono dalle tradizioni più varie. Ingegneria del software, supporto ad architetture convenzionali, parallele o distribuite, matematica, logica e intelligenza artificiale sono le tradizioni che hanno maggiormente influenzato la ricerca nel campo della progettazione dei linguaggi di programmazione.

In questa tesi prendo posizione a favore di un nuovo orientamento che nasce da motivazioni economiche, in particolare di economia del lavoro.

Motivazioni e obiettivi

La tesi ha due obiettivi ambiziosi: primo, dimostrare che esiste (almeno) una funzionalità generale non implementabile adeguatamente con gli attuali linguaggi di programmazione; secondo, individuare gli aspetti fondazionali responsabili di questo problema in modo da indicare un nuovo punto di partenza per la progettazione dei linguaggi di programmazione.

I linguaggi di programmazione più diffusi sono concepiti per uso generale e sono ritenuti sufficientemente versatili per poter scrivere le applicazioni più varie, sicuramente tutte quelle di uso comune. Un'idea generalmente accettata è che la realizzazione di queste applicazioni possa essere facilitata dallo sviluppo di librerie di componenti ma che non richieda un particolare supporto da parte del linguaggio di programmazione. Linguaggi come Java e C++ sono ritenuti sostanzialmente adeguati alle attuali esigenze di programmazione.

Parto dal modello dominante di programmazione orientata agli oggetti e pongo un Obiettivo di Adattabilità da raggiungere. Lo scopo è trovare nuovi modi di implementare estensioni non anticipate del comportamento senza riprogettare manualmente il codice esistente.

Lo scopo della tesi è dimostrare che non è possibile raggiungere l'Obiettivo di Adattabilità restando nell'ambito dei paradigmi di programmazione esistenti. In particolare: non è possibile raggiungerlo restando nell'ambito del paradigma di programmazione orientata agli oggetti; né è sufficiente fare un passo indietro e ripartire basandosi sul paradigma di programmazione imperativa, funzionale, dataflow o logica.

Scopo della tesi è anche determinare le caratteristiche necessarie e sufficienti che possano essere usate per ripartire nella definizione di un nuovo paradigma di programmazione che supporti l'Obiettivo di Adattabilità. A tal fine viene presentata una nuova unità funzionale non riducibile ai modelli di programmazione esistenti che rappresenta una soluzione a diversi problemi riscontrati negli altri paradigmi e che può essere sviluppata per supportare pienamente l'Obiettivo di Adattabilità

Una particolare enfasi viene posta sul fatto che gli attuali linguaggi obbligano il programmatore a sovraspecificare il programma in vari modi rendendo più difficile del necessario l'evoluzione del software e impedendo agli utenti delle applicazioni di adattare le funzionalità secondo le proprie esigenze.

Struttura della tesi

La tesi è organizzata in tre parti. La ricerca di una soluzione all'Obiettivo di Adattabilità definito nella prima parte fa da filo conduttore per sviluppare un ragionamento che porta ad escludere la possibilità di trovare una soluzione nell'ambito degli attuali paradigmi di programmazione.

Ogni argomento di ricerca viene posto in relazione alla tradizione in cui è nato: agli assunti, alle esigenze e agli obiettivi di chi lo porta avanti. Questo per capire cosa viene dato per acquisito e cosa invece viene considerato oggetto di ricerca e quindi sviluppabile.

Nella prima parte descrivo lo stato dell'arte nei linguaggi di programmazione e pongo l'Obiettivo di Adattabilità. Nella seconda parte provo a raggiungerlo restando nell'ambito del paradigma di programmazione orientata agli oggetti. Nella terza parte provo a raggiungerlo con gli altri paradigmi di programmazione.

Nella prima parte descrivo la piattaforma Java e il paradigma di programmazione orientata agli oggetti che rappresenta lo stato dell'arte nella progettazione dei linguaggi di programmazione. Poi indico un orientamento desiderabile per lo sviluppo delle applicazioni e fisso un Obiettivo di Adattabilità che i programmi devono raggiungere e che i linguaggi devono supportare.

Nella seconda parte analizzo gli orientamenti della ricerca che si muovono nell'ambito del paradigma di programmazione orientata agli oggetti. La ricerca di una soluzione all'Obiettivo di Adattabilità procede tentando di implementare l'operazione di differenziazione. Prendo in considerazione: i design pattern, i soggetti, gli aspetti, la programmazione basata su oggetti, i modelli basati su predicati, quelli ad attori e la riflessività. Inoltre analizzo i due aspetti più rilevanti della programmazione orientata agli oggetti: i meccanismi di condivisione e di instradamento.

Nella terza parte analizzo i quattro paradigmi di programmazione: imperativo, funzionale, dataflow e logico. La ricerca di una soluzione all'Obiettivo di Adattabilità prosegue tentando di dare una risposta alla domanda se posso ripartire la responsabilità della determinazione dei parametri attuali tra la chiamata a funzione e la funzione chiamata.

Contributi e limiti

In questa tesi presento un'operazione – la differenziazione – che conferisce ai programmi una determinata forma di comportamento adattabile. Ad esempio, se mentre uso un programma per leggere i *newsgroups* che permette di scegliere solo globalmente il font da usare per mostrare i messaggi, avverto l'esigenza di usare font diversi a seconda del newsgroup del messaggio, posso applicare la differenziazione per ottenere questo risultato nonostante che il programma sia stato progettato per non farlo.

I linguaggi di programmazione più usati per scrivere programmi sono concepiti per un uso generale; cioè sono ritenuti sufficientemente versatili per poter essere usati per scrivere i programmi più vari, sicuramente tutti quelli di uso comune. Faccio vedere che l'operazione di differenziazione non può essere supportata adeguatamente dagli attuali linguaggi di programmazione.

Mi propongo di fare due cose. Una è promuovere un determinato sviluppo degli attuali linguaggi ad oggetti in particolare Java. L'altra è cercare di comprendere da dove ricominciare la ricerca di un nuovo orientamento per la progettazione dei linguaggi di programmazione.

Faccio vedere che è possibile definire un modello computazionale con le seguenti caratteristiche. Le funzioni possono essere chiamate con un sottoinsieme dei parametri e provvedono a farsi calcolare quelli mancanti. L'algoritmo di ricerca/produzione dei parametri attuali può essere esteso esplicitamente e dinamicamente.

Inoltre faccio vedere che è possibile definire un sostituto dei puntatori che lascia aperta la determinazione dell'oggetto puntato e che è possibile sottrarre al programmatore la responsabilità di definire la struttura per rappresentare le entità complesse.

Sarebbe stato interessante confrontare con metodi formali i meccanismi di instradamento in modo da completare il lavoro già fatto con i meccanismi di condivisione.

Per confrontare i paradigmi di programmazione presento un modello di esecuzione basato sui flussi delle informazioni in particolare flusso della domanda e flusso dei dati. Sarebbe stato interessante definire una semantica del modello in modo da usare metodi formali per confrontare i modelli di esecuzione dei vari paradigmi.

La volontà di non presentare in questa sede un modello completo di programmazione rappresentativo del nuovo orientamento ha in parte complicato l'esposizione di alcune soluzioni proposte funzionali al raggiungimento dell'Obiettivo di Adattabilità rendendo difficile valutare la sufficienza del percorso di sviluppo del modello tracciato.

P A R T E I

STATO DELL'ARTE E OBIETTIVI

I linguaggi di programmazione più usati per scrivere programmi sono concepiti per un uso generale; cioè sono ritenuti sufficientemente versatili per poter essere usati per scrivere i programmi più vari, sicuramente tutti quelli di uso comune.

Una piattaforma come Java fornisce, oltre al semplice linguaggio, una vastissima quantità e varietà di librerie di componenti che facilitano la scrittura di altrettanti tipi di programmi. L'idea che si promuove e che viene generalmente accettata è che linguaggi come Java o C++ siano sostanzialmente adeguati alle attuali esigenze di programmazione; e che l'oggetto del contendere vada spostato sul terreno dei componenti.

Anche la ricerca nel campo dei linguaggi di programmazione rafforza questa idea. Il paradigma di programmazione orientata agli oggetti è considerato oggi un punto di riferimento. Esistono diversi gruppi di ricerca che esplorano possibili varianti di questo o quel meccanismo al fine di migliorare una qualche proprietà ingegnerestica (riusabilità, evolvibilità, …). Però tutte le proposte rimangono nell'ambito della programmazione orientata agli oggetti. Nessuno finora ha preso posizione sulla necessità di una alternativa. Abbiamo anzi assistito alla convergenza verso un modello ad oggetti anche di paradigmi – funzionale, dataflow e logico – molto lontani concettualmente.

Per mettere alla prova questa convinzione definisco una operazione – la differenziazione – che conferisce ai programmi una determinata forma di comportamento adattabile. Ad esempio, se mentre uso un programma per leggere i *newsgroups* che permette di scegliere solo globalmente il font da usare per mostrare i messaggi, avverto l'esigenza di usare font diversi a seconda del

newsgroup del messaggio, posso usare la differenziazione per ottenere questo risultato nonostante che il programma sia stato progettato per non farlo.

Poi mi domando se sia possibile supportare questa operazione in un qualche linguaggio di programmazione orientato agli oggetti. Dalla risposta dipenderà se posso cominciare a scrivere programmi che richiedono un comportamento adattabile con i linguaggi attuali oppure se devo prima pormi il problema di definire un nuovo linguaggio di programmazione adeguato. Una eventuale risposta negativa significherà anche che le varianti della programmazione orientata agli oggetti che si propongono di rendere più flessibili certi tipi di ristrutturazione delle classi hanno delle possibilità di riuscita superiormente limitate.

Questa prima parte è divisa in tre capitoli. Il primo descrive Java, il linguaggio attualmente dominante. Il secondo descrive il paradigma di programmazione orientata agli oggetti che rappresenta lo stato dell'arte nella progettazione dei linguaggi di programmazione e di cui Java è un esponente. Il terzo indica un orientamento desiderabile per lo sviluppo dei programmi e fissa un Obiettivo di Adattabilità che i programmi devono raggiungere e che pertanto i linguaggi devono supportare.

Capitolo 2

JAVA, IL MAINSTREAM

Java non è il migliore dei linguaggi possibili, ma è la più riuscita sintesi disponibile ora.

Java è stato inizialmente sviluppato per risolvere i problemi di sviluppo di applicazioni per le periferiche di consumo collegate in rete. I programmi Java sono indipendenti dalla piattaforma e dalla rete. La crescente popolarità di Internet ha valorizzato queste funzionalità – uniche allora – di Java e ha fatto da trampolino di lancio. L'impegno profuso nel migliorare l'efficienza che per molto tempo lo aveva penalizzato e nel facilitare lo sviluppo di applicazioni fornendo librerie sempre più complete e competitive, ha fatto il resto. I linguaggi – C++ e Delphi – che avrebbero potuto contrastare l'espansione di Java, sono invece rimasti ancorati alle singole piattaforme affidandosi alla rendita di posizione.

La piattaforma Java è composta dal linguaggio di programmazione vero e proprio, da un ricco insieme di librerie standard, da una macchina virtuale per eseguire i programmi e infine da una serie di strumenti (compilatore, debugger, generatore di chiavi e di certificati, …).

La piattaforma Java è disponibile in tre edizioni: Micro, Standard e Enterprise. A differenziarle sono la macchina virtuale e le librerie fornite di serie. Le tre edizioni coprono le esigenze di un mercato che va dai palmari ai server aziendali. Inoltre sono disponibili o in corso di sviluppo soluzioni complete che non richiedono un sistema operativo sottostante.

Tutte e quattro le componenti della piattaforma sono in evoluzione continua. Le librerie sono state in buona parte ridisegnate passando alla versione Java 2 e da allora ne vengono sempre aggiunte di nuove sotto forma di estensioni standard. La macchina virtuale è stata più volte riprogettata per fornire sempre maggiori prestazioni. Il linguaggio pur essendo rimasto comprensibilmente più stabile ha registrato dei miglioramenti; per il prossimo anno ad esempio è prevista l'aggiunta dei tipi parametrici. Infine anche gli strumenti di programmazione forniti sono stati migliorati significativamente e presto saranno affiancati da ambienti di sviluppo visuale diversificati per ciascuna edizione.

Il linguaggio Java

Java è un linguaggio concorrente orientato agli oggetti basato su classi; inizialmente è stato sviluppato da James Gosling [ArnGos96, GJS96]. Il nucleo di Java è imperativo e riprende la sintassi e la semantica del C e pertanto anche del C++.

L'unità fondamentale di programmazione in Java è la classe. Una classe è una descrizione di oggetti; le classi contengono dei metodi e la struttura degli oggetti, più i costruttori per realizzarli. Java supporta ereditarietà singola di implementazione e ereditarietà multipla di interfaccie. Una classe può estendere un'altra classe e può implementare diverse interfaccie. Una interfaccia dichiara i metodi supportati dalle classi che la implementano. Classi e interfaccie sono raggruppate in librerie (package) che hanno sia la funzione di definire spazi di nomi separati che quella di definire degli ambiti di visibilità per gli attributi che lo richiedono.

Java è un linguaggio fortemente tipato. I tipi sono divisi in due categorie: tipi primitivi e tipi riferimento. Una interfaccia è un tipo; i supertipi di una interfaccia sono le interfaccie che estende. Una classe è un tipo; i supertipi di una classe sono la classe che estende e le interfaccie che implementa più tutti i

supertipi di queste classi e interfaccie. Di conseguenza, i tipi di un oggetto sono la sua classe e tutti i suoi supertipi comprese le interfaccie.

Tra le funzionalità non legate né alla natura imperativa né a quella orientata agli oggetti si fanno notare per importanza il supporto alla concorrenza, la gestione delle anomalie e la gestione automatica della memoria.

Supporto alla concorrenza

Il supporto di Java alla programmazione concorrente consiste in un meccanismo per creare nuovi thread di esecuzione e in uno per gestire la sincronizzazione.

Ci sono due modi per creare nuovi thread di esecuzione. Uno è estendere la classe *java.lang.Thread* e ridefinire il metodo *run*; l'altro è implementare l'interfaccia *java.lang.Runnable* e definire il metodo *run*. In entrambi casi è necessario istanziare un Thread ed eseguire il metodo *start* che provvede a sua volta ad eseguire il metodo *run* su un thread concorrente.

Tutte le classi Java, in quanto estensioni della classe *java.lang.Object*, implementano il costrutto monitor. La parole chiave *synchronized* davanti alla definizione di un metodo o di un blocco ne garantisce l'esecuzione in mutua esclusione. I metodi *wait*, *notify* e *notifyAll* sono usati per implementare la politica di sincronizzazione tra i thread che accedono ad uno stesso oggetto.

Gestione delle anomalie

Le anomalie sono un meccanismo elegante per segnalare eccezioni direttamente anziché ricorrere a valori speciali di ritorno o ad effetti collaterali. In Java, ogni metodo dichiara i tipi di anomalie che può generare. Una anomalia è una classe che estende *java.lang.Exception* con attributi utili a gestirla L'istruzione *throw* genera una anomalia. Per intercettare una anomalia bisogna rinchiudere il codice che può generarla in un blocco *try* e farlo seguire da una *catch*, contenente il codice per gestirla, che intercetti quel tipo di anomalia. Si possono usare più *catch* in modo da gestire più tipi di anomalie e inoltre si può far seguire un blocco *finally* che viene eseguito comunque.

Gestione automatica della memoria

Il gestore automatico della memoria (garbage collector) è un meccanismo che elimina la necessità di liberare esplicitamente la memoria degli oggetti istanziati dinamicamente. Quando un oggetto non è referenziato da nessuno ad eccezione eventualmente di altri oggetti non referenziati, allora il suo spazio può essere riutilizzato.

La piattaforma Java 2 SE: la macchina virtuale

La Macchina Virtuale Java (JVM) è il componente responsabile della esecuzione e della portabilità dei programmi Java. La JVM non assume nessuna particolare tecnologia di implementazione; la piattaforma Java definisce le sue specifiche [LinYel96] e comprende delle implementazioni di riferimento ottimizzate per ciascuna architettura supportata e per ciascuna edizione della piattaforma stessa.

La JVM è una macchina astratta con un proprio set di istruzioni (*bytecode*), gestione della memoria, della concorrenza e della sicurezza. I sorgenti Java vengono compilati in Java *bytecode* e salvati nel formato di file *Java class*.

Una JVM può essere implementata come semplice interprete eventualmente affiancato da un compilatore just-in-time (JIT). Le attuali implementazioni di riferimento sono molto più sofisticate; prevedono un compilatore adattivo. Le applicazioni vengono lanciate con un semplice interprete e il codice viene analizzato durante l'esecuzione per individuare i colli di bottiglia (hot spots). La Java 2 Client VM compila queste porzioni di codice critiche servendosi dei risultati dell'analisi a tempo di esecuzione per determinare il modo migliore di ottimizzare il codice.

La piattaforma Java 2 SE: le principali librerie

In questa sezione descrivo brevemente le principali funzionalità fornite dalle librerie dell'edizione standard di Java. Il duplice scopo è dare un'idea della vastità di tipi di programmi che si possono scrivere facilmente (in modo incrementale).

In secondo luogo, per far comprendere anche quantitativamente il valore della portabilità dei programmi garantita da Java. Per una descrizione tecnica si rimanda alla documentazione ufficiale [Sun99, Sun99a].

Collezioni

La libreria di collezioni fornisce una architettura uniforme per rappresentare le collezioni e manipolarle indipendentemente dai dettagli implementativi. La libreria definisce interfaccie e diverse implementazioni delle più usate collezioni: liste, mappe, insiemi; inoltre fornisce alcuni algoritmi per manipolarle: ordinamento, ricerca, massimo e minimo, ...

Riflessività

La libreria Reflection fornisce introspezione sulle classi e gli oggetti della macchina virtuale in uso. Se la politica di sicurezza lo consente, è possibile ottenere un elenco dei campi, dei metodi e dei costruttori di una classe; inoltre i metodi possono essere invocati e i costruttori possono essere usati per istanziare nuovi oggetti.

Input/output

La libreria di input/output fornisce dei metodi per caricare e salvare una stream di dati.

Serializzazione

La libreria di serializzazione estende le classi di input/output con il supporto per gli oggetti. La libreria di serializzazione supporta la codifica di oggetti, e degli oggetti da essi raggiungibili, in stream di byte; e supporta la ricostruzione del grafo di un oggetto a partire da uno stream. La serializzazione viene usata per ottenere persistenza e nelle invocazioni di metodi remoti.

Interfaccia grafica

Le librerie relative alla interfaccia grafica forniscono un ricco insieme di componenti visuali, un modello di gestione degli eventi e delle funzioni di disegno. I componenti visuali possono cambiare, anche dinamicamente, aspetto

e comportamento in modo da seguire, ad esempio, le convenzioni del sistema operativo su cui si esegue il programma.

Interfaccia audio

La libreria audio permette di registrare, elaborare e riprodurre dati musicali nei più comuni formati.

Connettività

La libreria *java.net* fornisce delle classi per implementare applicazioni di rete. Permette di comunicare con un server su Internet o di implementarne uno. Sono supportati i protocolli TCP e UDP.

Programmazione distribuita

I programmi distribuiti in Java possono essere scritti con RMI o CORBA. La libreria RMI è più facile da usare e grazie alla libreria RMI-IIOP i programmi scritti con RMI possono usare il protocollo di comunicazione di CORBA e dialogare con clienti di ogni tipo.

Componenti JavaBeans

Definisce uno standard per realizzare componenti facilmente integrabili. Usando strumenti di costruzione di applicazioni che supportano i JavaBeans si possono combinare anche in modo visuale e interattivo questi componenti in applicazioni, applets o componenti composti.

Accesso ai database

La libreria JDBC permette a Java di accedere virtualmente a ogni sorgente di dati: dai database relazionali ai fogli di calcolo. JDBC fornisce delle API per eseguire istruzioni SQL e uno standard per scrivere dei driver di interfaccia con i principali DBMS. Sono disponibili i driver per tutti i principali DBMS oltre ad un driver che fa da ponte con lo standard ODBC.

Sicurezza

Il modello di sicurezza è basato su politiche, è facilmente configurabile e assicura un controllo degli accessi con una granularità fine. Quando viene caricato del codice, gli vengono assegnati dei permessi in base alla politica di sicurezza in vigore. Ogni permesso specifica quali accessi sono consentiti per una particolare risorsa. La politica di sicurezza definisce quali permessi sono disponibili per del codice in base alla provenienza e alle certificazioni che possiede.

Internazionalizzazione

La libreria di internazionalizzazione supporta lo sviluppo di applicazioni sensibili alla lingua e alle convenzioni culturali degli utenti. La libreria definisce uno standard per separare dal resto del programma gli elementi testuali e i dati con un formato dipendente dalla cultura. In questo modo si possono realizzare programmi multilingua e si può aggiungere il supporto ad una nuova lingua senza ricompilare il programma.

Interfaccia nativa

L'interfaccia nativa (JNI) è una interfaccia di programmazione standard per interfacciare Java con librerie native e applicazioni native con librerie Java. L'obiettivo principale è la compatibilità binaria tra le implementazioni della macchina virtuale Java in una data piattaforma.

Meccanismo di estensione

Le estensioni sono delle librerie che possono essere aggiunte alla piattaforma Java base. Il meccanismo di estensione consente alla macchina virtuale Java di usare le estensioni allo stesso modo delle classi base e di scaricarle automaticamente qualora non fossero disponibili in locale. Ogni programma Java può elencare le estensioni che richiede e la loro reperibilità.

Critica della piattaforma Java

Java e i sistemi operativi

Java attacca fortemente il concetto di sistema operativo in due aspetti: l'estensione e il ruolo predominante. La piattaforma Java è quasi un ambiente operativo che gira sui principali sistemi operativi attuali. È una piattaforma in buona parte autonoma e necessita solo di una piccola parte dei servizi forniti dagli attuali sistemi operativi. Vi è cioè una larga sovrapposizione tra le librerie fornite da un particolare sistema operativo e quelle che fanno parte della piattaforma Java.

Un sistema operativo è una entità separata e distinguibile sia dall'utente finale che dai programmatori. Per l'utente finale è una particolare interfaccia con relative convenzioni d'uso, ha delle funzionalità (multimediali, di navigazione su Internet, …) ed è un requisito per poter installare determinate applicazioni. Per un programmatore è un insieme di librerie (API) e di linee guida da usare e rispettare per poter sviluppare applicazioni compatibili.

La piattaforma Java pone le basi per riportare il concetto di sistema operativo ai suoi confini storici [Tan92] ma soprattutto pone le basi per ridimensionare l'importanza predominante che ha oggi un sistema operativo rispetto alle applicazioni. In prospettiva possiamo attenderci che il sistema operativo divenga intercambiabile come oggi lo è la macchina virtuale Java.

Java e gli altri linguaggi orientati agli oggetti

Prima del rilascio di Java, il C++ [Str91] era il linguaggio di riferimento. È difficile sostenere in assoluto la superiorità di Java come linguaggio rispetto al C++ o viceversa. A favore di Java ci possono essere il supporto alla concorrenza, l'uso dei riferimenti al posto dei puntatori e il garbage collector al posto della eliminazione esplicita degli oggetti dinamici. Il C++ per la concorrenza deve ricorrere a librerie esterne ma non è una caratteristica così strategica, e, d'altra parte, ha in più l'ereditarietà multipla e i template. Detto questo, le differenze tra Java inteso come linguaggio e il C++ sono insufficienti

a giustificare il ruolo dominante che ha assunto Java nel panorama dei linguaggi di programmazione.

A fare la differenza sono tutte quelle funzionalità tradizionalmente fornite dal sistema operativo o dall'hardware che invece si è scelto di integrare nella piattaforma Java. Per importanza cito il linguaggio degli eseguibili, il supporto alla concorrenza e tra le librerie l'interfaccia grafica e il supporto alla programmazione distribuita (su Internet). Avere queste funzionalità integrate al linguaggio rende le applicazioni interamente portabili. Anche limitandoci a considerare un determinato hardware e sistema operativo sottostante, continua ad essere una scelta vantaggiosa perché si tratta di librerie omogenee e scritte al livello di astrazione del linguaggio mentre i sistemi operativi più diffusi continuano ad avare API imperative. Quand'anche i sistemi operativi disponessero di librerie orientate agli oggetti sarebbe ancora preferibile integrare nel linguaggio queste funzionalità perché solo così si possono scrivere applicazioni che le usano senza ricorrere alla complicazione di design pattern per ridurre l'accoppiamento programma/librerie per rendere praticabile la portabilità. Acquisita l'idea di integrare le librerie al linguaggio (precisamente alla sua macchina virtuale), ci si può chiedere se sia conveniente definire le interfaccie e implementare solo degli adattatori (adapter pattern) che mappino le interfaccie portabili alle varie implementazioni dei sistemi operativi. La risposta varia da libreria a libreria e dipende da considerazioni ingegneristiche. Ad esempio, Java 1.0x adottava una libreria grafica mappata su quelle proprietarie; si è deciso di abbandonarla a favore di una implementazione più autonoma per diverse buone ragioni. Ricordo, tra le altre, l'esigenza di avere un look & feel consistente ed portabile, avere una gestione degli eventi completamente portabile e uniforme ed infine perché lo sforzo iniziale di scrivere librerie più grosse è ampiamente compensato dalla maggiore mantenibilità del codice, dall'assenza di fastidiosi bug di mappatura e perché semplifica la portabilità su nuove piattaforme.

Nella misura in cui si è diffusa la consapevolezza dell'importanza preminente della piattaforma Java rispetto al linguaggio, anche altri linguaggi di programmazione hanno imboccato la stessa strada. Anche il linguaggio C++ potrebbe benissimo adottare il bytecode e le librerie Java e in questo modo rimettersi in corsa riportando la competizione sul piano del linguaggio. La scelta, unanime per ora, fatta da chi implementa i linguaggi di programmazione, è stata di adottare non solo l'idea ma proprio la piattaforma Java. Sviluppare una nuova macchina virtuale e/o nuove librerie sarebbe molto più costoso e inoltre complicherebbe l'interoperabilità con i componenti già sviluppati per Java.

Capitolo 3

PROGRAMMAZIONE ORIENTATA AGLI OGGETTI

La programmazione orientata agli oggetti è diventata immensamente popolare negli ultimi anni, è ormai generalmente accettata come metodologia standard per lo sviluppo di applicazioni.

La programmazione orientata agli oggetti è superiore dal punto di vista dell'ingegneria del software. Non è che permetta di fare cose che i paradigmi di programmazione convenzionali non permettono, piuttosto consente di estendere le funzionalità di un programma aggiungendo del codice in casi in cui i paradigmi convenzionali richiedono di modificarlo.

La programmazione orientata agli oggetti non è un paradigma completo; di solito si appoggia ad un nucleo di linguaggio imperativo ma può essere applicata a tutti i paradigmi: funzionale, dataflow o logico.

Anatomia della programmazione orientata agli oggetti

La programmazione orientata agli oggetti enfatizza il concetto di oggetto. Un *oggetto* è una unità che ha uno stato e un comportamento. Lo stato è memorizzato in variabili d'istanza; il comportamento è definito da metodi. Nella maggior parte dei linguaggi orientati agli oggetti questi ultimi sono creati come istanze di classi. La classe definisce le variabili d'istanza e i metodi degli oggetti istanziati da quella classe.

La classe di un oggetto definisce la sua implementazione. Il tipo di un oggetto definisce la sua interfaccia cioè l'insieme di richieste a cui può rispondere. Un oggetto può avere più tipi e oggetti di classi differenti possono avere un tipo in

comune. Poiché una classe definisce le operazioni che un oggetto può eseguire, definisce anche il suo tipo.

I punti di forza dei linguaggi orientati agli oggetti sono il polimorfismo, l'incapsulamento e l'ereditarietà. Il *polimorfismo* permette di scrivere un cliente in termini delle operazioni di cui ha bisogno, piuttosto che degli oggetti specifici che manipolerà; ogni classe di oggetti che supporta le operazioni richieste può essere manipolata dal cliente senza modificare il cliente stesso. L'*incapsulamento* fa si che un cliente che usa i metodi di un oggetto possa conoscere e modificare di quell'oggetto solo quello che i metodi pubblici di quell'oggetto gli consentono. In questo modo lo stato di un oggetto può essere cambiato solo in accordo al protocollo che esso definisce. L'*ereditarietà* è un meccanismo per definire in modo incrementale un oggetto. Gli oggetti di una classe sono gli stessi di quelli di un'altra classe da cui ereditano a meno delle differenze specificate.

L'ereditarietà fra classi è un meccanismo di condivisione del codice che definisce l'implementazione di un oggetto in termini dell'implementazione di un altro. Mentre l'ereditarietà fra interfaccie (subtyping) stabilisce quando un oggetto può essere usato al posto di un altro.

Critica alla programmazione orientata agli oggetti

Un programma orientato agli oggetti consiste in una gerarchia di classi che riproduce una classificazione concettuale del dominio del problema che corrisponde a quella percepita dai suoi sviluppatori. La programmazione orientata agli oggetti, come paradigma, non prevede meccanismi di ristrutturazione delle classi e quindi incoraggia una visione oggettiva del mondo. I filosofi hanno a lungo dibattuto il problema se ci sia una singola tassonomia delle cose oggettivamente corretta (vedi [Tai96] per una rassegna). È facile constatare che una classificazione dipende in modo rilevante dalla cultura, dalle capacità e dall'esperienza di chi la fa; inoltre una stessa persona può percepire il mondo in differenti modi: in tempi diversi o volontariamente adottando diverse

prospettive o punti di vista. Non si conoscono regole universali per determinare quali proprietà usare come base per la classificazione degli oggetti; di conseguenza, il processo di classificazione non può essere fatto meccanicamente ma iterativamente alternando creatività e valutazione. Non si conosce neppure un criterio per stabilire se una data classificazione sia oggettivamente corretta; di conseguenza il processo dovrà non solo essere diretto ma anche terminato su base consensuale quando la classificazione trovata sia ritenuta abbastanza buona. Questi limiti attuali non escludono che possa esistere una soluzione oggettivamente corretta né che il processo iterativo converga a questa unica soluzione. Possiamo anzi assumere che per ogni dato problema si possa trovare una buona soluzione consensuale e considerare questa soluzione equivalente in pratica alla soluzione oggettivamente corretta. Il mondo reale e quello delle idee sono però in continua evoluzione e questo sposta sempre il problema. Queste considerazioni implicano che un buon progettista di software orientato agli oggetti debba essere sempre preparato al cambiamento; sempre intento a raggiungere iterativamente una soluzione abbastanza buona dopo l'altra.

L'osservazione che il paradigma orientato agli oggetti ha una capacità limitata di modellare il mondo reale perché al più può descrivere una buona istantanea di una realtà in divenire, non è realmente un problema perché i paradigmi alternativi attuali sono ancor meno competitivi e soprattutto perché l'evoluzione non è così ampia e frenetica da avere costi insostenibili.

L'esigenza di ristrutturare le classi di un programma è reale; tanto che la maggior parte delle varianti della programmazione orientata agli oggetti si propone di rendere più flessibili certi tipi di ristrutturazione. Il punto è quantificare la stabilità del dominio dei problemi che vengono affrontati oggi con i linguaggi orientati agli oggetti.

Capitolo 4

DOVE VOGLIAMO ANDARE OGGI?

“Tutte le nuove tecnologie si sviluppano in un contesto di tacita comprensione della natura e del lavoro dell’uomo. L’uso della tecnologia, a sua volta, porta a cambiamenti fondamentali in ciò che facciamo e in ultima istanza in ciò che significa essere umani. Incontriamo i quesiti profondi della progettazione quando ci rendiamo conto che nel progettare strumenti, stiamo progettando modi di essere.” [WinFlo87].

Le innovazioni tecnologiche possono riguardare il processo e/o il prodotto. Le innovazioni di processo possono indurre in diversi casi effetti occupazionali negativi mentre le innovazioni di prodotto sono accompagnate più frequentemente da effetti occupazionali positivi.

Un nuovo prodotto contribuisce alla fine del ciclo di vita dei prodotti che va a sostituire; favorisce una riduzione dei prezzi dei vecchi prodotti e una riduzione dell’occupazione nelle imprese che li producono. D’altra parte produce una domanda del nuovo prodotto che deve essere sostenuta aumentando la produzione e quindi aumentando l’impiego di lavoro. Inoltre un nuovo prodotto può creare una domanda prima inesistente.

L’adozione di tecnologie risparmiatrici di lavoro ha due tipi di effetti: un effetto di impatto sull’impiego dei fattori di produzione a parità di produzione finale nel settore di adozione e un effetto di compensazione legato al mutamento della scala di produzione.

Si distinguono tre effetti di compensazione: prezzo, reddito e moltiplicativi.

- Effetto prezzo. L'adozione di nuove tecnologie riduce i costi di produzione; quindi rende possibile una riduzione dei prezzi dei prodotti in una misura che dipende dalla forma di mercato del settore di adozione. La riduzione dei prezzi stimola la domanda dei beni (direttamente e indirettamente di altri beni) che porta ad un aumento della produzione e al conseguente riassorbimento della forza lavoro.

- Effetto reddito. L'adozione di nuove tecnologie determina una crescita dei redditi monetari sotto forma di profitti e di salari. I soggetti beneficiari di questa crescita, con un'autonoma decisione di spesa, possono contribuire al processo di accumulazione del capitale o alla crescita della domanda di beni di consumo. Il prevalere di quest'ultima scelta porta ad un aumento della produzione e al conseguente riassorbimento della forza lavoro.

- Effetti moltiplicativi. Questo effetto si manifesta quando il cambiamento tecnologico è incorporato in beni capitali. L'accresciuta produzione di tali beni determina una domanda addizionale di lavoro in questi settori. Quindi si ha una riduzione di lavoro nel settore che adotta le nuove tecnologie ed un aumento nei settori che producono i beni capitali.

Gli effetti di compensazione che ho descritto sono caratterizzati da sfasamenti spaziali e temporali. L'occupazione si riduce in alcuni settori o imprese ed aumenta in altri settori o imprese dislocate diversamente sul territorio. L'adozione di una tecnologia risparmiatrice di lavoro ha quasi sempre un effetto di impatto negativo per l'occupazione, solo a distanza di tempo gli effetti di compensazione possono ristabilire il livello occupazionale precedente o aumentarlo.

In economia del lavoro è ancora aperto il dibattito tra chi sostiene che la disoccupazione – conseguente l'adozione di tecnologie risparmiatrici di lavoro – venga più o meno automaticamente riassorbita in altri, eventualmente nuovi, settori e chi invece considera il processo inesorabilmente in espansione.

In entrambi i casi, i meccanismi di riassorbimento dell'occupazione descritti e le politiche atte a favorirli possono trarre maggior forza da tecnologie informatiche.

Una parte rilevante delle innovazioni tecnologiche risparmiatrici di lavoro è legata all'informatica. Dobbiamo progettare il software con la consapevolezza che la nostra opera contribuisce all'avanzamento di due processi in atto: l'automazione e la riorganizzazione.

Contribuiamo sia al processo di progressiva sostituzione di forza lavoro umana con forza lavoro macchina che qui chiamo genericamente automazione sia al processo di riassorbimento/creazione dell'occupazione che qui chiamo genericamente riorganizzazione. Ogni prodotto software contribuisce ad entrambi i processi sia pure in misura diversa.

Quali programmi possono favorire ovvero rendere più praticabili: flessibilità del lavoro, mobilità, lavoro interinale, incontro tra domanda e offerta di lavoro, incontro tra capitale e capacità imprenditoriali, flessibilità dell'impresa?

È convinzione diffusa che questi problemi non riguardano i linguaggi di programmazione.

La programmazione orientata agli oggetti può modellare adeguatamente una qualsiasi soluzione per ciascuno di questi problemi. Il punto è che non vi è una comprensione sufficientemente estesa e condivisa di questi problemi; non abbiamo soluzioni da specificare e tradurre in programmi.

Abbiamo due scelte o ci limitiamo a gestire le informazioni e a fornire strumenti generici di pianificazione e modellazione dei flussi informativi oppure cerchiamo di supportare la ricerca di soluzioni anche locali fornendo i programmi di un comportamento adattabile. Questa seconda soluzione consiste nel mettere ciascun soggetto coinvolto nei processi descritti nella condizione di adattare il

comportamento dei programmi che usa secondo le proprie convinzioni maturate in uno specifico ambito lavorativo.

Nel seguito di questo capitolo introduco una operazione – la differenziazione – che fornisce ai programmi una certa adattabilità di comportamento.

In questa sede non mi interessa né dare una definizione generale di comportamento adattabile né stabilire se l'operazione di differenziazione è sufficiente per supportare lo sviluppo di soluzioni autonome ai problemi posti. Definirò anzi l'operazione di differenziazione in una forma semplificata per facilitare lo sviluppo e la comprensione del ragionamento.

Lo scopo è stabilire se la programmazione orientata agli oggetti può supportare adeguatamente questa operazione e in che modo. Voglio mettere alla prova la convinzione che per risolvere problemi di adattabilità del comportamento siano sufficienti i linguaggi attuali come ad esempio Java o C++.

Esempio di riferimento

Per facilitare la comprensione del funzionamento dell'operazione di differenziazione e delle difficoltà che accompagnano la ricerca e la verifica di una sua implementazione, mi servo di un programma di esempio. Lo scopo è sceglierlo in modo che sia il più semplice possibile da comprendere e nel contempo abbastanza ampio da ammettere applicazioni significative dell'operazione di differenziazione. Ho scelto come esempio di riferimento il nucleo di un programma di gestione messaggi.

Faccio l'ipotesi che un programmatore ovvero una azienda di software decida di scrivere un programma per gestire i messaggi dei newsgroup distribuiti via Internet. Si intraprende il processo di analisi del problema e vengono determinati i requisiti che deve avere il programma. Seguono una fase di progettazione e una di implementazione. Nei due paragrafi che seguono descrivo le classi che fanno parte del nucleo del programma e la loro

implementazione in Java. Per semplificare ulteriormente l'esposizione mi limito a descrivere e ad implementare gli attributi delle classi rilevanti ai nostri fini. Il sorgente del programma risultante è compilabile ed eseguibile pertanto è adatto anche per verificare il funzionamento delle eventuali implementazioni dell'operazione di differenziazione.

Descrizione delle classi

La classe *Message* rappresenta un singolo messaggio. Per semplicità contiene solo il mittente e il soggetto di un messaggio con i relativi metodi di accesso.

La classe *Group* rappresenta un gruppo di messaggi. Contiene un nome e una lista di messaggi. Fornisce i metodi di accesso al nome e permette di aggiungere ed eliminare messaggi e di ottenere un iteratore sulla lista.

La classe *Prefs* contiene le preferenze relative alla visualizzazione del mittente e del soggetto di un messaggio con i relativi metodi di accesso.

La classe GroupsView permette di mostrare la lista dei gruppi. Definisce i metodi per aggiungere ed eliminare gruppi e fornisce i metodi di accesso per il gruppo attualmente selezionato.

La classe *HeadersView* permette di mostrare la lista delle intestazioni dei messaggi contenuti in un gruppo. Definisce i metodi di accesso per scegliere il gruppo e le preferenze di visualizzazione.

Infine la classe *NewsReader* assembla gli altri componenti e definisce il *main*. Il programma prevede un unico gruppo di messaggi e una sola vista per mostrarne le intestazioni. Permette di aggiungere ed eliminare messaggi e di scegliere quali attributi mostrare nella vista delle intestazioni.

Sorgente delle classi

```
public class Message {
    private String sender;
    private String subject;
```

```
    public Message(String sender, String subject) {
        setSender(sender);
        setSubject(subject);
    }

    public final void setSender(String sender) {
        this.sender = sender;
    }
    public final String getSender() {
        return sender;
    }

    public final void setSubject(String subject) {
        this.subject = subject;
    }
    public final String getSubject() {
        return subject;
    }
}

public class Group {
    private String name;
    private List messageList;

    public Group(String name) {
        setName(name);
        messageList = new ArrayList();
    }

    public final void setName(String name) {
        this.name = name;
    }
    public final String getName() {
        return name;
    }

    public void addMessage(Message msg) {
        messageList.add(msg);
    }
    public void removeMessage(Message msg) {
        messageList.remove(msg);
    }
    public ListIterator getMessageIterator() {
        return messageList.listIterator();
    }
}

public class Prefs {

    private boolean sender = true;

    private boolean subject = true;


    public final boolean isSenderVisible() {

        return sender;
```

```
    }
    public final void setSenderVisible(boolean val) {
        sender = val;
    }

    public final boolean isSubjectVisible() {
        return subject;
    }
    public final void setSubjectVisible(boolean val) {
        subject = val;
    }
}
public class GroupsView {
    private List groupList;
    private Group selectedGroup;

    public GroupsView() {
        groupList = new ArrayList();
        addGroup(new Group("default group"));
        setSelectedGroup(0);
    }

    public final void addGroup(Group group) {
        groupList.add(group);
    }
    public final void removeGroup(Group group) {
        groupList.remove(group);
    }

    public Group getSelectedGroup() {
        return selectedGroup;
    }
    public void setSelectedGroup(int index) {
        try {
            selectedGroup = (Group)groupList.get(index);
        } catch (IndexOutOfBoundsException e) {
```

```
        }
    }

    public void display() {
        System.out.println();
        System.out.println("Group list");

        Iterator i = groupList.listIterator();
        while (i.hasNext())
            System.out.println(((Group)i.next()).getName());
    }
}
public class HeadersView {
    private Group group;
    private Prefs prefs;

    public HeadersView(Group group, Prefs prefs) {
        setGroup(group);
        setPrefs(prefs);
    }

    public final void setGroup(Group group) {
        this.group = group;
    }
    public final Group getGroup() {
        return group;
    }

    public final void setPrefs(Prefs prefs) {
        this.prefs = prefs;
    }
    public final Prefs getPrefs() {
        return prefs;
    }

    public void display() {
        System.out.println();
```

```
        System.out.println("Group: "+group.getName());
        if (prefs.isSenderVisible())
            System.out.print("Sender      ");
        if (prefs.isSubjectVisible())
            System.out.print("Subject ");
        System.out.println();

        Iterator i = group.getMessageIterator();
        while (i.hasNext()) {
            Message msg = (Message)i.next();
            if (prefs.isSenderVisible())
                System.out.print(msg.getSender().concat("
").substring(0,12));
            if (prefs.isSubjectVisible())
                System.out.print(msg.getSubject());
            System.out.println();
        }
    }
}

public class NewsReader {
    public Prefs prefs;
    public GroupsView groupsView;
    public HeadersView headersView;

    public NewsReader() {
        prefs = new Prefs();
        groupsView = new GroupsView();
        headersView = new
HeadersView(groupsView.getSelectedGroup(), prefs);
    }

    public void selectGroup(int index) {
        groupsView.setSelectedGroup(index);
        headersView.setGroup(groupsView.getSelectedGroup());
    }
```

```
    public void addMessage(Message msg) {
        groupsView.getSelectedGroup().addMessage(msg);
    }

    // Il main oltre ad istanziare il programma contiene un
esempio d’uso. Per comodità ho inserito qui anche un esempio
d’uso dell’operazione di differenziazione che naturalmente
funziona solo assieme alle soluzioni che la implementano
descritte nella seconda parte della tesi.
    public static final void main(String[] arg) throws
NoSuchMethodException {
        NewsReader nr = new NewsReader();
        nr.groupsView.addGroup(new Group("gruppo 1"));
        nr.groupsView.addGroup(new Group("gruppo 2"));

        nr.selectGroup(1);
        nr.addMessage(new Message("Riccardo", "prova uno"));
        nr.addMessage(new Message("Clelia", "prova due"));
        nr.addMessage(new Message("Riccardo", "prova tre"));
        nr.selectGroup(2);
        nr.addMessage(new Message("Clelia", "prova quattro"));
        nr.addMessage(new Message("Riccardo", "prova
cinque"));

        nr.selectGroup(1);
        nr.groupsView.display();
        nr.headersView.display();
        System.out.println("*** nascondi subject");
        nr.prefs.setSubjectVisible(false);
        nr.headersView.display();
        nr.selectGroup(2);
        nr.headersView.display();

        System.out.println("*** differenzio Prefs rispetto al
gruppo visualizzato");
        nr.prefs.differentiate(
            nr.headersView,

    nr.headersView.getClass().getMethod("getGroup",null)
```

```
        );
        System.out.println("*** mostra subject");
        nr.prefs.setSubjectVisible(true);
        nr.headersView.display();
        nr.selectGroup(1);
        nr.headersView.display();
    }
}
```

Formulazione del problema

Voglio aggiungere ad ogni classe una operazione che ne contestualizzi il comportamento. Chiamo questa nuova operazione *differenziazione.* Un oggetto *a differenziato* da una variabile *b* si comporta come se avesse uno stato diverso per ogni valore assunto da *b.* Detto in altri termini, è come se ogni oggetto del tipo di *b* (ed effettivamente assegnabile) acquisisse, a seguito dell'operazione, una copia di *a* e tutte le operazioni su *a* venissero delegate alla copia posseduta dall'attuale oggetto referenziato da *b*.

Tornando all'esempio di riferimento, a tempo di esecuzione, ci sono le seguenti istanze delle classi definite: un GroupsView, un HeadersView, un Prefs, un NewsReader, enne Group e emme Message. Pertanto sono disponibili le seguenti variabili: groupsView, groupsView.selectedGroup, headersView, headersView.group e headersView.prefs, prefs. Di queste, quattro sono costanti: groupsView, headersView, headersView.prefs, prefs; inoltre le coppie headersView.prefs, prefs e groupsView.selectedGroup, headersView.group hanno sempre lo stesso valore.

L'operazione di differenziazione può essere applicata per ottenere ad esempio le seguenti modifiche al comportamento originale del programma.

- prefs differenziato da groupsView.selectedGroup – Ogni gruppo ha le proprie preferenze di visualizzazione.

- headersView differenziato da groupsView.selectedGroup – Ogni gruppo ha la propria vista di intestazioni.

- prefs differenziato da headersView – Ogni vista di intestazioni ha le proprie preferenze di visualizzazione.

- prefs differenziato da aMessage – Ogni messaggio ha le proprie preferenze di visualizzazione.

Analisi del problema

Per rendere più comprensibile l'esposizione uso i nomi dei riferimenti agli oggetti coinvolti nell'applicazione dell'operazione: prefs differenziato da selectedGroup.

Non so quanti e quali sono gli oggetti che contengono un riferimento a prefs; so che possono usare solo un sottoinsieme dell'interfaccia (eventualmente tutta) definita da Prefs.

Quando applico l'operazione di differenziazione a prefs, implicitamente, creo delle copie di prefs, ognuna associata ad un valore (oggetto) di selectedGroup.

La classe Group non contiene un attributo di tipo Prefs né di conseguenza i metodi per accedervi. Inoltre, l'insieme dei valori che può assumere selectedGroup è un sottoinsieme delle istanze di Group. Infine, la copia di prefs associata ad un gruppo viene usata solo se almeno una operazione su prefs viene eseguita quando selectedGroup fa riferimento a quel gruppo. Di conseguenza è sufficiente creare le copie di prefs man mano che servono.

Dopo la differenziazione, le variabili di prefs non vengono più usate e tutti i metodi vengono ridiretti su una copia di prefs. Chi può stabilire qual è l'attuale copia di prefs da usare?

I clienti di prefs vogliono usare dei metodi della classe Prefs sull'istanza che possiedono; dopo la differenziazione non sanno su quale istanza eseguirli.

Non posso aggiungere un parametro a tutti i metodi di prefs ed aspettarmi che gli oggetti che usano prefs si procurino selectedGroup e me lo passino. Né posso aggiungere a prefs un metodo setTargetPrefs ed aspettarmi che il metodo di assegnamento di selectedGroup lo chiami.

L'operazione di differenziazione coinvolge quattro soggetti: l'oggetto da differenziare, gli oggetti cliente che lo usano, gli oggetti destinatari delle copie e la variabile che indica il destinatario corrente. Per quanto detto, la responsabilità dell'operazione di differenziazione su un oggetto è difficilmente distribuibile agli altri soggetti coinvolti. Non conviene ripartire la responsabilità anche per una questione di molteplicità degli oggetti clienti e destinatari.

L'operazione di differenziazione ha effetti sul comportamento dei clienti dell'oggetto differenziato. Se nell'esempio di riferimento differenzio le preferenze rispetto al gruppo selezionato gli effetti dell'operazione si manifestano sui clienti delle preferenze. Il programma principale, dopo l'operazione, mostra e permette di cambiare le preferenze di visualizzazione del gruppo selezionato. Analogamente la finestra di visualizzazione delle intestazioni dei messaggi mostra gli attributi dei messaggi indicati nelle preferenze di visualizzazione del gruppo selezionato.

Sorgente adattato manualmente

Il sorgente che segue contiene tutti i cambiamenti che ho dovuto fare al sorgente originale perché ogni gruppo abbia le proprie preferenze di visualizzazione.

```
public class Group {
    private Prefs prefs; //added

    public Group(String name) {
        …
        prefs = new Prefs(); //added
    }

    //added
```

```
    public final void setPrefs(Prefs prefs) {
        this.prefs = prefs;
    }
    //added
    public final Prefs getPrefs() {
        return prefs;
    }
}

public class HeadersView {
    //private Prefs prefs; removed

    public HeadersView(Group group/*, Prefs prefs*/) {
        …
        //setPrefs(prefs); removed
    }

/* removed
    public final void setPrefs(Prefs prefs) {
        this.prefs = prefs;
    }
    public final Prefs getPrefs() {
        return prefs;
    }
*/
    public void display() {
        Prefs prefs = group.getPrefs(); //added
        …
        }
    }
}

public class NewsReader {
    //public Prefs prefs; removed

    public NewsReader() {
        //prefs = new Prefs(); removed
        …
        headersView = new
HeadersView(groupsView.getSelectedGroup()/*, prefs*/);
    }

// added
    public final Prefs getPrefs() {
        return groupsView.getSelectedGroup().getPrefs();
    }

    public static final void main(String[] arg) throws
NoSuchMethodException {
        …
        nr.getPrefs().setSubjectVisible(false); //changed
        …
        nr.getPrefs().setSubjectVisible(true); //changed
        …
    }
}
```

Caratteristiche di una soluzione adattiva

La soluzione deve rendere possibile la separazione degli aspetti algoritmici di un programma da quelli riguardanti la differenziazione dei campi di un oggetto. Deve poter essere usata dai programmatori per definire una configurazione iniziale del programma e durante l'esecuzione dagli utenti per adattare il comportamento del programma alle proprie esigenze.

L'operazione di differenziazione deve essere implementata esplicitamente ad un qualche livello accessibile al programma ma deve poter essere implementata una volta per tutte.

La soluzione non deve richiedere ai programmatori nessuno sforzo specifico di programmazione. Al massimo è concesso che richieda di definire le classi estendendo quelle date, e che preveda anche uno stile di programmazione da adottare.

L'operazione di differenziazione deve operare una trasformazione persistente del programma. Il risultato delle operazioni di differenziazione deve sopravvivere sia alla chiusura e successiva riesecuzione del programma sia all'introduzione di nuove versioni dello stesso.

Per valutare la bontà delle soluzioni trovate uso come termine di paragone il programma su misura che avrei scritto se avessi deciso fin dalla fase di progettazione di fornire le funzionalità aggiunte dalle operazioni di differenziazione eseguite.

L'*Obiettivo di Adattabilità* consiste nel trovare una implementazione dell'operazione di differenziazione che soddisfi tutti i requisiti esposti.

Conclusioni

In questo capitolo pongo il problema di supportare lo sviluppo di applicazioni che devono operare in domini non sufficientemente compresi. Generalmente questo non è considerato un problema da affrontare a livello dei linguaggi di programmazione. L'introduzione dell'operazione di differenziazione ha proprio lo scopo di verificare quanto sia vero questo assunto.

L'operazione di differenziazione ha una applicabilità generale, in questo capitolo l'ho presentata in una forma limitata per facilitare la sua implementazione. Vi sono diversi modi per aumentarne l'espressività:

- Definire una operazione simmetrica per uniformare un comportamento differenziato. In questo modo la differenziazione diventa reversibile.

- Oltre a ridirigere un metodo su una copia di un oggetto differenziato, deve essere possibile anche distribuire l'esecuzione su tutte le copie magari limitatamente ad alcune operazioni.

- Per poter essere usata dagli utenti del programma bisogna definire una rappresentazione visuale. Si può aggiungere uno strumento – il differenziatore – che quando è attivo permette di scegliere l'oggetto da differenziare e la variabile di differenziazione selezionando con il puntatore gli elementi dell'interfaccia del programma e le parti dei documenti aperti.

- La variabile di differenziazione deve poter essere scelta anche al di fuori degli oggetti definiti dal programma. Si possono aggiungere ad esempio contenitori generici di liste in modo da differenziare un elemento del programma rispetto al valore selezionato nella lista.

Nella seconda parte della tesi prendo in considerazione diverse varianti della programmazione orientata agli oggetti alla ricerca di una soluzione dell'Obiettivo di Adattabilità.

P A R T E I I

TENDENZE INTERNE AL PARADIGMA OO

Nella prima parte è stato descritto il linguaggio di programmazione dominante e ho posto un Obiettivo di Adattabilità da raggiungere. In questa parte vengono presi in considerazione gli orientamenti della ricerca che si muovono nell'ambito del paradigma di programmazione orientata agli oggetti che è lo stesso dei linguaggi più diffusi. Le tendenze attuali sono esposte in ordine crescente di quantità di cambiamenti da apportare al modello dominante; e quindi danno anche una misura della crescente difficoltà con cui si può raggiungere l'Obiettivo.

Dapprima prendo in considerazione tendenze nate nell'ambito dell'ingegneria del software. Le ipotesi di soluzione descritte sono per nulla o poco intrusive. Si propongono di addestrare meglio i progettisti (design pattern), di affiancare un linguaggio di composizione a quello orientato agli oggetti (programmazione orientata ai soggetti) o al massimo propongono di affiancare altre entità alle classi (programmazione orientata agli aspetti).

Poi mi chiedo se la scelta di un modello concettuale basato su oggetti anziché su classi possa dare la flessibilità richiesta. Gli oggetti intuitivamente privilegiano l'individualità rispetto all'uniformità rappresentata dalle classi. Inoltre sono spesso accompagnati da meccanismi di condivisione diversi, alternativi all'ereditarietà.

Poi rivolgo l'attenzione a modelli che aumentano la flessibilità del meccanismo di instradamento facendo uso di predicati o di un modello ad attori. Se il problema è dovuto al fatto che un oggetto è un aggregato di attributi posso provare ad aggirarlo permettendo esplicitamente di cambiare l'instradamento dei messaggi prolungandolo fin dove è necessario.

Infine con il modello riflessivo provo a spostare all'interno del linguaggio tutte le operazioni di cui dispone un programmatore per scrivere e modificare i programmi. Se le modifiche da apportare ad un programma per raggiungere l'Obiettivo di Adattabilità non richiedono inventiva, devono essere esprimibili come sequenze di operazioni riflessive.

Questa seconda parte della tesi è divisa in capitoli come segue. Il primo descrive i design pattern. Il secondo la programmazione orientata ai soggetti. Il terzo descrive la programmazione orientata agli aspetti facendo riferimento in particolare al linguaggio AspectJ. Il quarto capitolo introduce la programmazione basata su oggetti in Self e Kevo in modo da confrontare nel quinto capitolo i meccanismi di condivisione. Il sesto e il settimo descrivono rispettivamente i modelli basati su predicati e il modello ad attori che vengono confrontati nell'ottavo capitolo dal punto di vista del meccanismo di instradamento. Il nono capitolo è dedicato alla riflessività. Infine il decimo alle conclusioni.

Capitolo 5

DESIGN PATTERN

La flessibilità e la riusabilità sono due degli obiettivi della programmazione orientata agli oggetti. I principi del paradigma object-oriented promuovono questi obiettivi ma da soli non li garantiscono; molto è lasciato alla capacità e all'esperienza di chi progetta il software. Non basta organizzare i programmi in classi per renderli flessibili. Le classi sono legate tra loro da gerarchie di condivisione e di composizione e anche le chiamate a funzione determinano delle collaborazioni e quindi delle dipendenze tra classi.

L'uso dei design pattern nella programmazione orientata agli oggetti nasce dalla tradizione dell'ingegneria del software. I design pattern descrivono delle soluzioni che non richiedono speciali funzionalità nei linguaggi di programmazione per essere usate. I design pattern richiedono solo una maggiore conoscenza da parte di chi progetta il software.

Un design pattern è uno schema di soluzione che ha dimostrato la propria flessibilità in un certo contesto e il cui riuso può dare analoghe garanzie. L'idea è catturare le esperienze di progettazione in forma di catalogo usabile da altri progettisti. Le più famose raccolte di design pattern si possono trovare in [GHJV95, BMRSS96, MRB98].

Ogni design pattern fornisce una qualche forma di flessibilità che consente di realizzare alcuni tipi di cambiamento in modo non intrusivo. In genere, per ottenere questo risultato, l'aspetto che può variare viene incapsulato e vengono introdotti dei livelli di indirezione addizionali nelle classi che lo usano.

Anatomia di un design pattern

Un design pattern è una descrizione di classi e di oggetti comunicanti strutturati in modo da risolvere un problema generale di progettazione in un particolare contesto [GHJV95]. I design pattern devono essere catalogati in modo uniforme e devono avere un nome in modo da formare un vocabolario di progettazione. Devono identificare le classi e le istanze che partecipano, il loro ruolo, le collaborazioni e la distribuzione di responsabilità. Devono descrivere quando possono essere applicati e le conseguenze del loro uso in termini di flessibilità ed estendibilità.

Tipi di design pattern

I design pattern variano per livello di astrazione, granularità ed entità coinvolte (classi/oggetti). Per facilitare la consultazione dei cataloghi è conveniente raggrupparli; in accordo a [GHJV95] si possono distinguere tre tipi di design pattern: creazionali, strutturali e comportamentali.

I design pattern creazionali astraggono il processo di istanziazione. Incapsulano la conoscenza di quali classi concrete vengono usate e nascondono il come vengono istanziate e composte. Rendono flessibile il cosa viene creato, chi, come e quando lo crea. I design pattern strutturali descrivono come classi e oggetti possono essere composti per formare strutture più grandi. I design pattern comportamentali si occupano di algoritmi e di assegnamento di responsabilità tra oggetti.

Principi di riusabilità

La chiave per massimizzare la riusabilità è anticipare i nuovi requisiti e i cambiamenti a quelli attuali e progettare il sistema in modo che possa evolvere facilmente nelle direzioni anticipate. Ogni design pattern permette a qualche aspetto della struttura del sistema di evolvere indipendentemente dagli altri aspetti. Le linee guida che hanno portato alla definizione dei design pattern e che

sono alla base di uno stile di programmazione orientato ai pattern sono le seguenti.

Gli oggetti vanno usati solo in termini di una loro interfaccia non della classe che li implementa. In questo modo il cliente rimane all'oscuro del tipo specifico degli oggetti che usa e anche delle classi che li implementano, gli basta assicurarsi che gli oggetti aderiscano all'interfaccia che si aspetta. Anche la creazione degli oggetti va fatta indirettamente in modo da non legarsi ad una particolare implementazione.

Per riusare una funzionalità ci si deve servire di preferenza della composizione di oggetti piuttosto che dell'ereditarietà di classi. La composizione di oggetti è definita dinamicamente a tempo di esecuzione assegnando dei puntatori; ogni oggetto può essere sostituito a patto che il nuovo implementi l'interfaccia richiesta. Uno svantaggio è che mentre gli attributi ereditati confluiscono negli oggetti che li ereditano, la composizione di oggetti mantiene separati gli attributi pertanto richiede più oggetti e di conseguenza il funzionamento del sistema dipende dalle loro interconnessioni. Un altro svantaggio è che l'interfaccia degli oggetti componenti non va automaticamente ad estendere quella dell'oggetto contenitore ma quest'ultimo deve ridirigere opportunamente i metodi che intende delegare. L'ereditarietà estende automaticamente l'interfaccia delle sottoclassi ma è definita staticamente dal programmatore e in genere non è possibile cambiare durante l'esecuzione la parte di implementazione ereditata. Inoltre le gerarchie di ereditarietà sono legate dall'implementazione e il cambiamento di una classe può richiedere dei cambiamenti nelle sottoclassi.

Critica ai design pattern

I design pattern non richiedono né aggiungono funzionalità ai linguaggi di programmazione orientati agli oggetti. I design pattern si limitano a promuovere un uso diverso degli attuali linguaggi.

L’assenza di supporto diretto da parte dei linguaggi di programmazione comporta una implementazione sparsa dei design pattern con conseguente perdita di località di espressione. *Località di espressione* significa che tutti gli elementi devono essere visibili insieme, e possono essere immediatamente riconosciuti e compresi. Per riconoscere un design pattern sparso, ogni parte deve essere prima riconosciuta localmente e poi le sue relazioni e l'intero pattern possono essere riconosciuti.

La dispersione del codice relativo ad una funzionalità è un fenomeno negativo intrinseco alla programmazione orientata agli oggetti; i design pattern non sono capaci di contrastarlo ma almeno, in genere, rendono locali alcune funzionalità che altrimenti sarebbero disperse e lasciano dispersi solo dei punti di indirezione.

Il problema dell’anticipazione

I design pattern devono essere applicati prima che si manifesti la loro necessità per una particolare evoluzione del programma. Questo si verifica perché deve essere aggiunto del codice nelle classi base per poter isolare delle funzionalità e renderle estendibili.

Il problema della perdita di identità

Ogni volta che un singolo oggetto individuato nella fase di progettazione (design object) viene rappresentato da più oggetti nell’implementazione (implementation object) si verifica il problema della perdita d’identità. La singola identità di un oggetto viene persa perché lo stato e/o il comportamento di quello che deve apparire come un singolo oggetto è diviso, nell’implementazione, in diversi oggetti ognuno inevitabilmente con la propria identità. Il problema è che l’esistenza di identità separate rende possibili delle violazioni anche involontarie alle assunzioni sul protocollo di interazione degli oggetti. In alcuni design pattern è possibile individuare un oggetto che rappresenti all’esterno l’identità ma non è la regola.

Contributi all'Obiettivo di Adattabilità

Ogni oggetto per implementare l'operazione di differenziazione, deve esibire due comportamenti: uno prima di essere differenziato e l'altro dopo. Il design pattern State permette di raggiungere esattamente questo scopo. Devo aggiungere ad ogni classe un campo contenente lo stato dell'oggetto e devo riscrivere tutti i metodi in modo che si limitino a delegare l'operazione all'omonimo metodo definito nello stato corrente. Devo poi definire per ogni classe l'interfaccia degli stati e le due classi che la implementano: una con il comportamento differenziato e l'altra con il corpo dei metodi originali. Vediamo come dovrebbe essere riscritta la classe Prefs per applicare questo schema di soluzione.

Sorgente della soluzione

```
public class Prefs {
    protected boolean sender = true;
    protected boolean subject = true;
    private PrefsState state = new PrefsOriginalState();

    public final boolean isSenderVisible() {
        return state.isSenderVisible(this);
    }
    public final void setSenderVisible(boolean val) {
        state.setSenderVisible(this, val);
    }
    …
    public void differentiate(…) {
        …
        state = new PrefsDifferentiatedState();
        …
    }
}

public interface PrefsState {
    public boolean isSenderVisible(Prefs self);
    public void setSenderVisible(Prefs self, boolean val);
    public boolean isSubjectVisible(Prefs self);
```

```
    public void setSubjectVisible(Prefs self, boolean val);
}

public class PrefsOriginalState implements PrefsState {
    public final boolean isSenderVisible(Prefs self) {
        return self.sender;
    }
    public final void setSenderVisible(Prefs self, boolean
val) {
        self.sender = val;
    }
    …
}

public class PrefsDifferentiatedState {
    …
}
```

Limiti della soluzione

La soluzione è chiaramente del tutto inadeguata. Per ogni classe del programma originale ne devo definire manualmente (seppure in modo meccanico) ben quattro. Ho parzialmente compromesso l'incapsulamento della classe Prefs e ho severamente complicato la possibilità di definire delle sottoclassi. Usando un semplice blocco *if* all'interno di ogni metodo per selezionare i due diversi comportamenti avrei perlomeno ridotto i problemi derivanti dalla proliferazione degli oggetti.

Conclusioni

I design pattern hanno contribuito in modo rilevante ad aumentare la flessibilità e la riusabilità del software. Per loro natura però, non rappresentano un passo avanti nella progettazione dei linguaggi di programmazione; semmai favoriscono la comprensione delle rigidità degli attuali linguaggi e possono suggerire direzioni di cambiamento.

Ad esempio, il design pattern Visitor attacca l'incapsulamento, evidenziando la necessità di avere degli scope trasversali alla gerarchia di ereditarietà. I design

pattern State e Strategy mostrano la necessità di avere oggetti che cambiano il proprio comportamento e che la ridirezione di un metodo è una approssimazione insoddisfacente della delega.

Al fine del raggiungimento dell'Obiettivo di Adattabilità, i design pattern, non forniscono nessun aiuto perché sono delle soluzioni generali che devono essere specificate per il singolo problema non è possibile individuare un design pattern o una loro combinazione ed applicarli una volta per tutte alle classi astratte; devono essere applicati classe per classe.

Il riconoscimento che esistono dei pattern inerenti alle soluzioni di alcuni problemi di evoluzione e riusabilità suggerisce l'idea di supportare direttamente nei linguaggi un tipo di programmazione orientata ai design pattern. La programmazione soggettiva e quella per aspetti presentate nei prossimi due capitoli si muovono entrambe in questa direzione.

Capitolo 6

PROGRAMMAZIONE ORIENTATA AI SOGGETTI

Anche la programmazione orientata ai soggetti nasce dalla tradizione dell'ingegneria del software applicata ai linguaggi di programmazione orientati agli oggetti. Diversamente dai design pattern che interessano unicamente i progettisti, fornisce soluzioni che coinvolgono l'ambiente di programmazione. Anche in questo approccio, i linguaggi di programmazione non vengono cambiati.

L'obiettivo dichiarato è complementare la programmazione orientata agli oggetti risolvendo alcuni problemi di coordinazione tra progettisti che sorgono quando si sviluppano grossi sistemi o suite di applicazioni integrate o che interagiscono. La programmazione soggettiva aggiunge delle funzionalità ai linguaggi orientati agli oggetti per favorire certi tipi di composizione dei programmi. Per una introduzione più approfondita si può fare riferimento a [OHBS94].

Permette di creare estensioni e diverse configurazioni di programmi senza modificare il codice sorgente. Facilita lo sviluppo di applicazioni da parte di più team. Permette di decentrare lo sviluppo delle classi. Indirizza il problema di supportare diverse prospettive sul dominio del problema. In particolare, semplifica il codice di molti design pattern.

Anatomia di un soggetto

Un *soggetto* è una collezione di classi o parti di classi la cui gerarchia modella il proprio dominio da una certa prospettiva. Un soggetto può essere una applicazione o una parte incompleta che deve essere composta con altri soggetti per produrre una applicazione autonoma. La programmazione orientata ai

soggetti supporta la costruzione di sistemi orientati agli oggetti a partire da soggetti. I soggetti vengono uniti usando delle regole di composizione.

Il processo di composizione stabilisce delle *corrispondenze* fra elementi di programma come classi e metodi e deriva elementi del programma composto combinando gli elementi corrispondenti. Le particolari corrispondenze usate e i dettagli per combinarle sono definiti dalle *regole di composizione* usate nella *specifica di composizione*. La programmazione orientata ai soggetti affianca cioè un linguaggio per scrivere le specifiche di composizione ad un linguaggio orientato agli oggetti.

Un ambiente di programmazione per un linguaggio orientato agli oggetti che supporti la programmazione orientata ai soggetti deve comprendere un compilatore di soggetti, un linguaggio di composizione e un compositore di soggetti. In alternativa se sono disponibili tutti i sorgenti è sufficiente un linguaggio di composizione, un compositore di soggetti e un compilatore tradizionale. In [OHBS94 e KOHK96] viene descritta una implementazione per supportare la programmazione orientata ai soggetti in C++.

Un compilatore di soggetti (*subjectifier*) esegue delle operazioni specifiche per il linguaggio sorgente e necessarie per creare un soggetto binario (*binary subject*) che possa essere composto in seguito. Le operazioni consistono in trasformazioni del codice sorgente che interessano i punti che contengono creazione/eliminazione di oggetti, accesso a variabili di istanza e chiamate a funzioni. Lo scopo è permettere di legare questi punti a delle definizioni di classi dopo la compilazione. Un soggetto binario contiene anche delle informazioni riguardanti la gerarchia di classi che fanno parte del soggetto. Il programmatore può scegliere quali parti delle interfaccie delle classi esporre ad altri soggetti. Il risultato della compilazione di un soggetto è indipendente dal linguaggio sorgente e può essere combinato anche con soggetti scritti in altri linguaggi.

Un compositore di soggetti crea un soggetto eseguibile a partire da una collezione di soggetti e una specifica di composizione. Il principale lavoro svolto

dal compositore è creare codice e tabelle per controllare l'instradamento dei messaggi tra i soggetti.

Critica della programmazione a soggetti

I design pattern sono una soluzione progettuale e implementativa che richiede di scrivere il codice in un certo modo. I soggetti esistono solo a tempo di progettazione poi vengono composti in una soluzione tradizionale. I soggetti permettono di separare il codice sorgente e sviluppare le parti autonomamente ma non richiedono di modificarlo per supportare questa possibilità. Prendiamo l'esempio classico di un programma di disegno di forme geometriche. Viene definita una gerarchia di forme disegnabili e ogni classe che implementa una forma concreta definisce un proprio metodo di disegno. La funzione che disegna l'intero documento risulta così dispersa in tante classi. L'applicazione del design pattern *Visitor* permette di concentrare in un'unica nuova classe – il visitor – tutti i metodi di disegno sparsi nelle diverse classi lasciando al loro posto un semplice metodo – accept – che ridirige la chiamata sul metodo di disegno specifico per quella classe contenuto nel visitor. La soluzione a soggetti si limita a separare in due soggetti la soluzione classica: uno con tutti i metodi di disegno e l'altro con tutto il resto.

Alcuni design pattern come State e Strategy descrivono dei comportamenti dinamici che spesso devono restare tali. Non è possibile sostituirli con una composizione statica di soggetti se non in quei (rari) casi in cui i design pattern citati sono usati per rappresentare dei comportamenti alternativi tra cui si può scegliere a tempo di compilazione. La consapevolezza di questo limite e dell'importanza dei comportamenti dinamici modellati dai design pattern si è tradotta per ora in una dichiarazione di intenti e di indirizzo. Come possibili soluzioni per supportarli sono stati indicati alcuni meccanismi che permettono di cambiare dinamicamente il tipo di una classe.

L'obiettivo principale della programmazione orientata ai soggetti è consentire uno sviluppo delle applicazioni decentrato e dal punto di vista scelto autonomamente da ciascun programmatore. Negli ultimi mesi è stato posto al centro dell'attenzione l'ambiente di programmazione; il nuovo obiettivo è dare a ciascun programmatore la possibilità di accedere al proprio sorgente scegliendo di volta in volta il punto di vista più adatto. Un sorgente testuale è statico e impone il proprio unico punto di vista. Un ambiente di programmazione può generare dinamicamente dei punti di vista sul programma corrispondenti agli aspetti che interessano.

Conclusioni

La programmazione orientata ai soggetti è ancora in fase di sviluppo preliminare; gli strumenti per applicarla non sono pubblicamente disponibili. Il contributo che può dare al raggiungimento dell'Obiettivo di Adattabilità è lo stesso della programmazione ad aspetti e pertanto verrà illustrato nel prossimo capitolo ad essa dedicato.

La programmazione orientata ai soggetti viene considerata una generalizzazione della programmazione ad aspetti dai ricercatori di entrambe le parti. Poiché la maggiore generalità consiste oggi solo nell'avere indicato obiettivi più ambiziosi da raggiungere, ho preferito riservare alla programmazione ad aspetti il capitolo conclusivo sulle tendenze di sviluppo della programmazione orientata agli oggetti nate nell'ambito dell'ingegneria del software. Pur essendo anche la programmazione ad aspetti molto giovane ha già saputo tradurre in meccanismi concreti e un linguaggio di programmazione usabile alcune delle proprie idee.

Capitolo 7

PROGRAMMAZIONE ORIENTATA AGLI ASPETTI

AspectJ

Anche la programmazione orientata agli aspetti nasce dalla tradizione dell'ingegneria del software applicata ai linguaggi di programmazione orientati agli oggetti. A differenza dei design pattern e della programmazione orientata ai soggetti coinvolge direttamente i linguaggi. I benefici principali che offre sono riduzione della dimensione del codice e maggiore facilità di sviluppo e mantenimento.

Diverse funzionalità dei programmi coinvolgono o richiedono la collaborazione di gruppi di oggetti; cioè per loro natura tagliano trasversalmente (cross-cut) la divisione primaria in oggetti fatta dalla programmazione orientata agli oggetti. Per questa ragione queste funzionalità sono difficili da modularizzare usando solo una divisione in oggetti. Gli aspetti sono un nuovo tipo di costrutto di programmazione che facilita l'implementazione, la composizione e il riuso delle funzionalità che tagliano trasversalmente il sistema.

Gli aspetti permettono di localizzare l'implementazione di alcuni design pattern anziché disperdere i campi e i metodi di questi pattern in diverse classi. Ad esempio, possono trarre vantaggio dall'uso di aspetti il protocollo di interazione fra un insieme di oggetti che collaborano all'esecuzione di un task più grande. Un altro esempio classico è dato dal supporto alla gestione delle eccezioni che per sua natura è sparso in tutto il sistema negli oggetti che possono lanciare delle eccezioni.

Gli aspetti tagliano trasversalmente la modularità delle classi. Un aspetto può influenzare l'implementazione di diverse classi (o di diversi metodi in una stessa

classe). Un tessitore di aspetti (aspect weaver) provvede a combinare automaticamente gli aspetti con le classi e può essere implementato come interprete, compilatore o preprocessore.

Anatomia di un aspetto in AspectJ

AspectJ è una estensione orientata agli aspetti per il linguaggio Java; l'attuale implementazione consiste in un precompilatore. Il linguaggio è in corso di sviluppo presso la Xerox Corporation, informazioni aggiornate si possono trovare nel sito ufficiale [Xer99]. Per una introduzione più generale alla programmazione orientata agli aspetti si consiglia [KLMM97].

Un aspetto, come una normale classe Java, ha degli attributi con tanto di modificatori di accesso e in più può dichiarare due tipi di *tessiture* (wheave): inserzioni (advice) e introduzioni (introduction).

Inserzioni e introduzioni contengono un *designatore* (crosscut) che specifica quali parti del programma (classi, metodi, variabili) sono interessate dalla dichiarazione. Un designatore può denotare esplicitamente una singola parte o implicitamente più parti facendo uso di wildcard; inoltre i designatori possono essere composti (and, or, not). L'attuale implementazione di AspectJ non prevede la possibilità di specificare l'ordine di composizione di più tessiture in una parte.

Le dichiarazioni di *introduzione* aggiungono nuovi attributi alle classi designate. Le dichiarazioni di *inserzione* aggiungono blocchi di codice in metodi e costruttori che già esistono nelle classi designate. Le inserzioni possono essere fatte all'inizio (*before*) o alla fine (*after*) di un metodo oppure possono intercettare le eccezioni (*catch*) o essere eseguite alla fine del metodo indipendentemente dalle eccezioni eventualmente lanciate (*finally*). Le eventuali variabili locali definite in una inserzione hanno come scope l'inserzione stessa e possono essere usate per mantenere un contesto fra le parti di una inserzione.

All'interno delle introduzioni e delle inserzioni possono essere usate le seguenti variabili riservate: thisObject, thisResult, thisResultObject e thisJoinPoint. La variabile thisObject è un riferimento all'oggetto corrente. La variabile thisResult è legata all'eventuale valore di ritorno del metodo corrente e thisResultObject è il suo tipo. La variabile thisJoinPoint è un oggetto che contiene le seguenti informazioni sul metodo corrente: il nome della classe e del metodo, i parametri attuali e il loro tipo.

AspectJ supporta due stili di programmazione orientata agli aspetti: tessitura statica e tessitura dinamica. Nella *tessitura statica* gli aspetti non vengono istanziati e le tessiture (inserzioni, introduzioni) vengono incorporate staticamente dal compilatore o preprocessore nelle classi designate; e quindi a tempo di esecuzione esistono solo oggetti regolari. Tutto il codice che lega gli oggetti deve essere isolato in aspetti. Nella *tessitura dinamica* gli aspetti vengono istanziati e possono essere aggiunti e rimossi ad uno o più oggetti a tempo di esecuzione.

Critica della programmazione ad aspetti

Gli aspetti attaccano su due fronti il paradigma di programmazione orientata agli oggetti: la modularità data dagli oggetti e l'incapsulamento. Sostengono cioè l'insufficienza degli oggetti come elementi di modularità e la necessità di estendere trasversalmente alla divisione in oggetti il concetto di incapsulamento.

Entrambi gli attacchi sono reali solo se si considerano gli aspetti delle entità del linguaggio come fa AspectJ. Tale interpretazione è necessaria solo per gli aspetti che richiedono una tessitura dinamica.

Incapsulamento

Il principio di incapsulamento stabilisce che gli attributi privati di un oggetto siano accessibili solo ai metodi dello stesso oggetto. In diversi linguaggi l'accesso privilegiato agli attributi privati di un oggetto viene consentito a tutti gli oggetti istanziati da una stessa classe. Inoltre di solito vengono forniti dei modificatori di controllo d'accesso in modo da poter definire attributi accessibili anche dagli

oggetti delle sottoclassi (protected), da tutti gli oggetti delle classi di un pacchetto o infine da tutti gli oggetti (public). La direttiva public esclude l'incapsulamento e dovrebbe essere usata solo per i metodi che devono far parte dell'interfaccia. Il linguaggio C++ dà anche la possibilità di elencare in una classe quali altri metodi o classi abbiano un accesso privilegiato (friend). La direttiva friend è trasversale e selettiva ma limitata agli usi che possono essere anticipati.

Gli aspetti rappresentano un attacco strutturato al concetto di incapsulamento. Un aspetto si comporta come parte delle classi che arricchisce e pertanto ha accesso anche ai loro attributi privati. Se si cambia una classe, può essere necessario cambiare anche gli aspetti che la arricchiscono; ma non è necessario cambiare le altre classi arricchite dagli stessi aspetti.

Limiti dei designatori

I designatori permettono di individuare i punti (crosscut) del sorgente dove andare ad aggiungere delle tessiture. Le entità che possono essere designate sono solo le classi e i metodi. Non è possibile designare ad esempio i punti del programma che invocano un metodo o accedono ad una variabile di una certa classe.

Contributi all'Obiettivo di Adattabilità

Ogni oggetto per implementare l'operazione di differenziazione, deve esibire due comportamenti uno prima di essere differenziato e l'altro dopo. La programmazione orientata agli aspetti mi consente di aggiungere degli attributi in ogni classe e di inserire del codice in ogni metodo. Quest'ultima caratteristica mi permette di definire un blocco *if* all'inizio di ogni metodo con il comportamento che devono esibire i metodi dopo che l'oggetto è stato differenziato.

L'operazione di differenziazione prevede quattro partecipanti. Di questi solo due possono essere designati con il linguaggio ad aspetti AspectJ: l'oggetto che viene differenziato e i metodi che possono fornire una variabile di differenziazione. Scelgo, per semplicità, di implementare l'intera operazione nell'oggetto che viene

differenziato; in questo modo ogni classe definisce la propria operazione di differenziazione e ne è interamente responsabile.

La soluzione proposta prevede l'introduzione in ogni classe di tre variabili e una funzione di differenziazione, e l'inserzione all'inizio di ogni metodo di un blocco *if*. Le tre variabili contengono: il metodo da usare per ottenere il destinatario attuale, l'oggetto su cui applicare il metodo e un mapping dagli oggetti destinatari alle copie differenziate. Il metodo di differenziazione si limita ad inizializzare le tre variabili; il blocco *if* ridirige il metodo sulla copia dell'oggetto associata all'attuale destinatario se è già stata eseguita l'operazione di differenziazione altrimenti lascia proseguire l'esecuzione locale. Le copie vengono create al volo la prima volta che vengono usate.

Sorgente della soluzione

```
public aspect Adaptable {
    crosscut classDefCut(): *;
    crosscut exclusionMethodsCut(): void differentiate(..) |
Object getTarget() | Object clone();
    crosscut voidMethodDefCut(): !static void *(..) &
!exclusionMethodsCut();
    crosscut methodDefCut(): !static * *(..) &
!exclusionMethodsCut();

    introduction classDefCut() {
        private Map map;
        private Object selectorParent;
        private Method getSelector;

        public void differentiate(Object selectorParent,
Method getSelector) {
            map = new HashMap();
            this.selectorParent = selectorParent;
            this.getSelector = getSelector;
        }

        private Object getTarget() {
            try {
                Object selector =
getSelector.invoke(selectorParent, null);
                Object target = map.get(selector);
                if (target == null) {
                    Map temp = thisObject.map;
                    thisObject.map = null;
                    target = thisObject.clone();
                    thisObject.map = temp;
                    map.put(selector, target);
```

```
                }
                return target;
            } catch (Exception e) {
                System.out.println("getTarget() error"+e);
                return null;
            }
        }
    }

    static advice voidMethodDefCut() & classDefCut() {
        before {
            if (thisObject.map != null) {
                    ((thisClass)thisObject.getTarget())
                         .thisMethod(thisParameters);
                return;
            }
        }
    }

    static advice (methodDefCut() & !voidMethodDefCut()) &
classDefCut() {
        before {
            if (thisObject.map != null) {
                return ((thisClass)thisObject.getTarget())
                    .thisMethod(thisParameters);
            }
        }
    }
}
```

Limiti della soluzione

L'implementazione limita l'applicabilità dell'operazione di differenziazione alla disponibilità di un metodo senza parametri per ottenere la variabile di differenziazione. Avrei potuto estendere l'applicabilità richiedendo come parametro direttamente la variabile ma in questo modo avrei violato l'incapsulamento dello stato di un oggetto. E comunque anche così non avrei raggiunto le variabili definite localmente nei metodi. Per poter applicare la differenziazione anche rispetto ad una variabile locale ho bisogno di estendere le operazioni di assegnamento su questa variabile con un prolungamento che porti il nuovo valore nell'oggetto differenziato invocando ad esempio un metodo setTarget.

Il valore corrente della variabile di differenziazione può essere ottenuto dall'oggetto differenziato con un apposito metodo *getTarget* oppure può essere fornito esplicitamente con un metodo setTarget. La getTarget, da sola,

garantisce la compatibilità all'indietro con tutti i clienti che, per forza di cose, non passano esplicitamente la variabile di differenziazione. La setTarget, da sola, permette di scrivere dei nuovi clienti che passano esplicitamente il parametro aggiunto e di usare come variabili di differenziazione anche quelle locali ai metodi. Una soluzione più generale deve integrare entrambe le possibilità tenendo conto del fatto che la setTarget pone dei problemi di interferenze tra chiamate.

Il limite più grave di questa soluzione è però un altro: non si tratta di una soluzione persistente. Durante l'esecuzione del programma posso applicare l'operazione di differenziazione ma al termine del programma perdo tutte le modifiche fatte. Potrei mantenere una lista delle operazioni di differenziazione eseguite, salvarla in fase di chiusura e caricarla al successivo riavvio. Purtroppo l'operazione di differenziazione viene eseguita su oggetti istanziati non su classi.

Inoltre il risultato dell'applicazione dell'operazione di differenziazione non assomiglia minimamente alla soluzione che può essere scritta da un programmatore. Questo pone due problemi: primo un programmatore che decidesse di supportare le funzionalità aggiunte eseguendo una sequenza di operazioni di differenziazione si troverebbe a dover fare un grosso lavoro di riprogettazione e implementazione. Secondo il programma risultante dovrebbe risolvere anche problemi di compatibilità con gli eventuali documenti prodotti dagli utenti con le funzionalità aggiuntive.

Il maggior merito di questa soluzione è che necessita solo di una ricompilazione dei programmi per aggiungere l'operazione di differenziazione e non fa nessuna assunzione sul modo in cui sono scritti.

Conclusioni

La programmazione orientata agli aspetti rappresenta un passo avanti significativo dal punto di vista dell'ingegneria del software. Gli aspetti aumentano la località di espressione e riducono la quantità di codice da scrivere.

Gli aspetti permettono di concentrare in un unico modulo tutto il codice e i dati di una funzionalità del programma che altrimenti verrebbe dispersa tra le classi. In fase di precompilazione, tutti gli attributi definiti in un aspetto vengono distribuiti nuovamente nel sorgente del programma; ognuno viene aggiunto nei punti designati.

Gli aspetti sono interessanti soprattutto come manipolatori statici del sorgente nel senso che ho appena detto. L'attuale implementazione di AspectJ [Xer99] ha ancora una capacità limitata di designare punti del sorgente dove fare le aggiunte; inoltre il supporto alla tessitura dinamica degli aspetti ha una applicabilità limitata ed è destinato ad essere sostituito nella prossima versione maggiore del linguaggio.

Con gli aspetti è possibile definire una implementazione dell'operazione di differenziazione che ha il merito di funzionare semplicemente ricompilando i programmi. Purtroppo questa implementazione non produce risultati persistenti e quindi non può essere considerata una soluzione dell'Obiettivo di Adattabilità.

Capitolo 8

PROGRAMMAZIONE BASATA SU OGGETTI

Self e Kevo

Le idee alla base della programmazione class-based e object-based si possono fare risalire agli inizi degli anni sessanta; diciamo rispettivamente con Simula nel 1967 e Sketchpad nel 1963. I linguaggi object-based vengono usati in studi fondazionali e programmazione esplorativa ma anche in alcuni ambiti accessibili ad un pubblico più vasto come i palmari (NewtonScript) e i browser (JavaScript). I sostenitori della programmazione basata su oggetti attribuiscono al proprio modello maggiore concretezza, semplicità concettuale e flessibilità. Non di meno il paradigma dominante è quello basato su classi e ad oltre trent'anni di distanza nessun linguaggio object-based può essere considerato un serio antagonista di linguaggi come C++ o Java.

Rappresentazione delle astrazioni

È stato osservato in [Tai96] che la distinzione tra sistemi basati su classi e sistemi basati su oggetti riflette una disputa filosofica sulla rappresentazione delle astrazioni.

Un gruppo di oggetti appartiene ad una stessa categoria se hanno le stesse proprietà. Pertanto, le categorie di oggetti sono definite dalle proprietà comuni ad un gruppo di oggetti. Nuove categorie possono essere definite in termini di altre categorie se le nuove hanno delle proprietà in comune con le vecchie.

Il filosofo Ludwig Wittgenstein ha osservato che è difficile dire in anticipo esattamente quali caratteristiche sono essenziali per un concetto. Vi sono dei concetti semplici ma estremamente difficili da definire in termini di proprietà condivise ad esempio il concetto di gioco o di opera d'arte. Per questo

Wittgenstein propone di definire un concetto non con delle proprietà condivise ma con una famiglia di somiglianze.

La classificazione è un processo induttivo che procede da una collezione di istanze verso le categorie cioè dal basso verso l'alto. I linguaggi basati su classi al contrario promuovono uno sviluppo delle classi dall'alto verso il basso.

I modelli basati su classi hanno concettualmente il problema della regressione infinita delle metaclassi. Una classe è un oggetto di un tipo che contiene un template per oggetti di un altro tipo e nessun oggetto è autosufficiente.

Anatomia di un linguaggio basato su oggetti

Il punto di partenza è il riconoscimento che le nozioni basate su classi non hanno bisogno di essere assunte, possono essere emulate dagli oggetti e combinate in modi più flessibili. In un linguaggio basato su oggetti non esistono classi ma solo oggetti detti anche prototipi. Gli oggetti vengono creati copiando altri oggetti. Per una rassegna dei principali linguaggi basati su oggetti si rimanda a [Smi94]. In questo capitolo introduco due linguaggi di programmazione rappresentativi dei due tipi di meccanismo di condivisione alternativi all'ereditarietà.

Self: un linguaggio basato su delega

Henry Lieberman ha proposto in [Lie86] di sostituire l'ereditarietà con la delega, avanzando l'idea che la condivisione fra oggetti possa essere realizzata mediante inoltro di messaggi.

Self è un linguaggio di programmazione sviluppato da David Ungar [UngSmi91]; è basato su oggetti, ha delega dinamica e variabili attive. La sintassi del linguaggio si ispira allo Smalltalk. Ogni oggetto può essere usato come istanza o servire come deposito per attributi condivisi. Nuovi oggetti vengono definiti usando altri oggetti come parenti, esplicitando quindi la condivisione del codice e dei dati. Questo meccanismo, chiamato delega, consente ad un oggetto di

inoltrare ad altri oggetti i messaggi che non sa come gestire localmente. In Self la delega è dinamica nel senso che è possibile cambiare l'oggetto parente. L'accesso ad un campo non è distinguibile dalla chiamata ad un metodo e il campo può essere sostituito da un metodo che lo calcola, senza modificare il codice degli oggetti cliente.

I linguaggi basati su delega pongono eccessiva enfasi sulla condivisione; l'attenzione del programmatore deve essere spesso rivolta ad aspetti puramente implementativi.

Kevo: un linguaggio basato su concatenazione

Kevo è un linguaggio di programmazione sviluppato da Antero Taivalsaari [Tai92, Tai93]; è basato su oggetti autosufficienti, è concorrente e riflessivo. La sintassi del linguaggio si ispira al Forth. Kevo differisce dalla maggior parte dei linguaggi orientati agli oggetti perché non enfatizza il meccanismo di condivisione; la gerarchia di condivisione viene anzi gestita automaticamente dal sistema. I programmi sono organizzati attorno alla gerarchia di composizione; gli oggetti sono autosufficienti e direttamente manipolabili. Nuovi oggetti possono essere creati per copia, la modificazione incrementale è ottenuta grazie a delle operazioni che consentono di manipolare gli oggetti. Per rendere possibili modifiche agli oggetti a livello di gruppo, Kevo usa la nozione di famiglia di copie (clone family). Una *famiglia di copie* è un gruppo di oggetti che hanno la stessa struttura e comportamento. Per ogni oggetto Kevo mantiene automaticamente la corrispondente famiglia di copie.

Kevo supporta luno stile di programmazione per esempi. Si parte definendo oggetti concreti e quando ci si accorge che un nuovo oggetto deve avere qualcosa in comune con uno già definito si copiano le caratteristiche desiderate.

Kevo supporta due tipi di gerarchie di oggetti. La gerarchia di composizione (part-whole hierarchy) che viene gestita dall'utente e la gerarchia di condivisione (clone family hierarchy) che è gestita automaticamente dal sistema e tiene traccia

delle parti in comune tra gli oggetti. Le due gerarchie sono ortogonali e descrivono esattamente lo stesso insieme di oggetti.

Kevo è progettato per essere un sistema interattivo visuale. Senza l'ambiente di programmazione visuale non è possibile apprezzare l'autosufficienza degli oggetti. La metafora usata è quella della scrivania. Gli oggetti sono rappresentati da cartelle contenenti gli attributi. Il programmatore può navigare nel sistema aprendo e chiudendo cartelle e può eseguire operazioni come creare, editare, eliminare, rinominare, copiare e muovere attributi. Il sistema mantiene traccia delle operazioni di copia (clone family) e quando si esegue una operazione si può scegliere se eseguirla sul singolo oggetto o anche sulle copie.

Critica della programmazione basata su oggetti

Gran parte dei meriti e dei limiti descritti in letteratura e attribuiti all'uno o all'altro modello sono in realtà da attribuire a proprietà indipendenti e applicabili ad entrambi i modelli e che solo per ragioni storiche si trovano prevalentemente adottate nell'uno o nell'altro. Proprietà come le operazioni sugli attributi, l'uniformità di trattamento degli attributi e l'ereditarietà dinamica sono presenti nella maggior parte dei linguaggi object-based ma non in quelli class-based. Queste proprietà sono una naturale estensione dei linguaggi basati su oggetti ma non sono intrinsecamente legate ad essi come dimostrano alcuni linguaggi basati su classi (Hybrid).

Operazioni sugli attributi temporalmente illimitate

Nei linguaggi basati su oggetti le operazioni sugli attributi sono spesso esplicite e ammesse in qualsiasi momento (operazioni sui record). Nei linguaggi basati su classi in genere sono implicite e limitate alla definizione della classe (sintattiche). I linguaggi basati su oggetti rimangono tali anche scegliendo di consentire solo operazioni sintattiche di aggiunta di campi e di aggiunta e ridefinizione di metodi nell'ambito della definizione di nuovi oggetti. Analogamente le operazioni sui

record possono essere aggiunte ai linguaggi basati su classi senza snaturarli come dimostra ad esempio Hybrid.

Uniformità di trattamento degli attributi

Molti linguaggi basati su oggetti (ad esempio Self) gestiscono in modo uniforme campi e metodi. In questo modo, l'accesso ad un campo non è distinguibile dalla chiamata ad un metodo e il campo può essere sostituito da un metodo che lo calcola, senza modificare il codice degli oggetti cliente.

Nei linguaggi in cui l'accesso ad un attributo è interpretabile come invio di un messaggio basta variare l'interpretazione del messaggio a seconda o meno della presenza di un metodo di accesso al campo. In linguaggi come Java in cui è più appropriata una interpretazione di accesso tramite scostamento rispetto la base della memoria dove è allocato l'oggetto, si può fare in modo che il compilatore definisca implicitamente (al pari del costruttore di default) i metodi di lettura e scrittura degli attributi e interpreti gli accessi e gli assegnamenti come zucchero sintattico. In questo modo le definizioni di nuovi metodi di accesso sarebbero automaticamente utilizzati da tutte le classi cliente senza modifiche né ricompilazioni; inoltre, in presenza di un compilatore just-in-time non si avrebbe alcun calo di prestazioni.

Attributi condivisi

Un attributo si dice condiviso se è accessibile e modificabile da più oggetti. Le modifiche, se si tratta di un campo, sono osservabili da tutti gli oggetti che possono accedere al campo.

Il modello basato su delega supporta in modo naturale gli attributi condivisi in particolare garantisce la non escludibilità della condivisione. Per avere analoghe garanzie strutturali nei modelli basati su ereditarietà e su concatenazione è necessario aggiungere una parola chiave (static) per distinguere definizioni di campi condivisi da campi di istanza.

Linguaggi come Java permettono di definire campi e metodi statici. I campi condivisi sono equivalenti ai campi statici di Java ad eccezione del fatto che non necessitano dell'indicazione della classe come risolutore di scope. I metodi statici di Java vengono eseguiti in uno scope limitato ai soli attributi statici e con *this* slegato.

Ereditarietà dinamica

Questa proprietà consiste nella possibilità di definire e modificare a tempo di esecuzione la gerarchia di condivisione; è presente in Self e nella maggior parte dei linguaggi basati su delega dei messaggi.

L'espressività aggiunta viene in buona parte catturata dai design pattern Strategy e State che sono utilizzabili anche in linguaggi basati su classi servendosi di ridirezione di messaggi anziché delega. La sua esclusione equivale alla determinazione preprogrammata e anticipata dei percorsi di condivisione per gruppo.

L'ereditarietà dinamica può essere aggiunta ad un linguaggio basato su classi definendo un modo per cambiare la superclasse della classe di un oggetto e modificando l'implementazione del meccanismo di condivisione. Hybrid è uno dei pochissimi linguaggi basati su classi che supporta l'ereditarietà dinamica. In genere i linguaggi basati su classi non forniscono nessun meccanismo di specializzazione dinamica del comportamento.

Dominio dell'allocazione dinamica

I linguaggi basati su classi forniscono l'operatore new, mentre i linguaggi basati su oggetti usano clone. Le due operazioni sono diverse; in particolare il dominio di new è limitato alle entità definite staticamente cioè le classi; invece la clone, limitandosi a copiare, è applicabile a tutti gli oggetti compresi quelli allocati dinamicamente. Per uniformare le due operazioni di allocazione dinamica si possono fare due scelte: o affiancare una clone alla new nei linguaggi basati su classi (vedi Java) oppure si può limitare il dominio di applicabilità della clone dei linguaggi basati su oggetti.

Costruttori

I costruttori hanno il compito di inizializzare lo stato degli oggetti allocati dinamicamente. Il costruttore viene invocato dalla operazione di allocazione dinamica e dà garanzie di uniformità degli oggetti creati. Di solito solo i linguaggi basati su classi supportano i costruttori ma è solo una scelta di comodo per quanto ragionevole. Un costruttore è opportuno per le classi e per i prototipi in modo da inizializzare i campi; nel caso di una clone di un oggetto allocato dinamicamente si deve poter scegliere se copiare semplicemente lo stato o applicare anche un costruttore.

Istanziabilità

La maggior parte dei linguaggi orientati agli oggetti distingue sintatticamente le entità istanziabili (con new/clone) da quelle che partecipano unicamente alla gerarchia di condivisione. Le entità non istanziabili prendono solitamente i nomi di, rispettivamente, classi astratte, tratti, aspetti. Una classe astratta dichiara metodi che non definisce. Tratti ed aspetti accedono ad attributi non definiti.

Conclusioni

I linguaggi basati su oggetti presentano spesso delle funzionalità poco diffuse tra i linguaggi basati su classi. Le operazioni sugli attributi, l'uniformità di trattamento degli attributi e l'ereditarietà dinamica, per elencare solo le principali, sono una naturale estensione dei linguaggi basati su oggetti ma non sono intrinsecamente legate ad essi. Sono funzionalità indipendenti e applicabili ad entrambi i modelli e solo per ragioni storiche si trovano prevalentemente adottate nell'uno o nell'altro. A distinguere la programmazione basata su oggetti da quella basata su classi resta il meccanismo di condivisione che verrà confrontato nel prossimo capitolo.

L'ereditarietà dinamica fornita da Self, per la sua possibile rilevanza al fine del raggiungimento dell'Obiettivo di Adattabilità verrà analizzata assieme agli altri meccanismi di specializzazione dinamica.

Capitolo 9

MECCANISMI DI CONDIVISIONE

L'ereditarietà è il meccanismo di condivisione più diffuso ed è spesso indicata come la proprietà distintiva che differenzia la programmazione orientata agli oggetti da altri paradigmi. Il termine ereditarietà è spesso usato come sinonimo di meccanismo di condivisione.

Sono stati fatti alcuni tentativi di definire una tassonomia per i meccanismi di condivisione esistenti [SLU88, Tai93]. L'intento dichiarato in questi scritti è stato di classificare tutte le possibili varianti; un solo tentativo è stato fatto per isolare i meccanismi da proprietà accidentali o non esclusive e confrontarli con metodi formali in modo da individuare le reali differenze [Sol99a]. Inoltre, in letteratura è rappresentata prevalentemente la contrapposizione tra meccanismi basati su classi e meccanismi basati su oggetti; questi ultimi identificati con il meccanismo di delega. Solo di recente è stata compresa l'originalità e la rilevanza della concatenazione.

Il meccanismo di instradamento dei messaggi fa da supporto al meccanismo di condivisione e al meccanismo di specializzazione dinamica del comportamento. Diversamente dalla maggior parte delle tassonomie non considero la dinamicità una variante della condivisione; la considero separatamente come origine del meccanismo di specializzazione dinamica. L'essere statico o dinamico è una proprietà ortogonale alle altre come già riconosciuto in [SLU88] pertanto la scelta è legittima e come vedremo vantaggiosa sia per confrontare i meccanismi di condivisione sia per confrontare i meccanismi di specializzazione dinamica.

Ogni meccanismo di condivisione definisce un modello concettuale con una propria terminologia ed una propria metafora di riferimento. Altre differenze sono osservabili nel modello di implementazione e nella rappresentazione;

ancora una volta è importante chiedersi quali differenze siano inerenti ai modelli e quali invece siano motivate solo da ragioni storiche e possano essere superate.

Un meccanismo di condivisione comprende un meccanismo di conformità ed uno di allocazione. In questo capitolo vengono presi in considerazione tre meccanismi di conformità: ereditarietà, delega e concatenazione, e due meccanismi di allocazione: istanziazione e clonazione. L'istanziazione e l'ereditarietà sono implementati nella maggior parte dei linguaggi basati su classi come ad esempio Java; la clonazione assieme a delega o a concatenazione sono implementati prevalentemente in linguaggi basati su oggetti come rispettivamente Self e Kevo.

Il confronto non può essere fatto direttamente sui linguaggi citati perché oltre al meccanismo che vogliamo analizzare possiedono altre proprietà indipendenti che condizionerebbero l'esito. Il confronto deve essere fatto alla pari isolando il meccanismo di condivisione: eliminando ovvero vincolando le altre funzionalità in modo da ottenere tre nuclei di linguaggio sostanzialmente identici a meno del meccanismo che ci interessa.

Il capitolo è organizzato come segue. Prima descrivo i tre meccanismi di condivisione. Poi definisco informalmente i tre linguaggi da confrontare e ne dimostro l'equivalenza facendo riferimento alle definizioni formali in [Sol99a]. Infine considero le conseguenze della equivalenza nei modelli di implementazione e di rappresentazione dei linguaggi.

Anatomia di un meccanismo di condivisione

Un meccanismo di condivisione ha due aspetti fondamentali: conformità e allocazione. Non è possibile definire un aspetto in termini dell'altro [SLU88]. Il meccanismo di conformità consente di avere una stessa informazione accessibile e modificabile in più contesti. Il meccanismo di allocazione consente di ottenere, data una informazione, due informazioni accessibili e modificabili separatamente.

I due aspetti di un meccanismo di condivisione possono essere meglio precisati rendendo le definizioni ortogonali. Si può definire un meccanismo dinamico e un meccanismo strutturale. Il primo rende disponibile una informazione in contesti diversi; il secondo permette di stabilire se l'informazione resa disponibile debba intendersi la stessa o distinta.

Meccanismi di conformità: ereditarietà, delega, concatenazione

Un meccanismo di conformità consente di avere una stessa informazione accessibile e modificabile in più contesti.

La definizione data vincola gli estremi dello spazio delle soluzioni. Da un lato possiamo mantenere una unica informazione, dall'altro possiamo replicarla per ogni accesso. Nel primo caso le richieste di accesso e modifica dovranno essere inoltrate dai contesti interessati all'informazione e la modifica verrà eseguita su quell'unica informazione (ereditarietà, delega). Nel secondo caso le richieste di accesso e modifica verranno eseguite localmente ma la modifica dovrà coinvolgere l'informazione in tutti i contesti in cui è replicata (concatenazione).

Ereditarietà

Una classe definisce l'insieme dei metodi e dei campi condivisi da tutte le istanze della classe e un insieme di campi specifici per ogni istanza. Le definizioni vengono ereditate da tutte le sottoclassi; in questo senso l'ereditarietà può essere definita un meccanismo di modificazione incrementale in presenza di un riferimento all'oggetto (this) legato in base al contesto (late-bound).

L'ereditarietà non può implementare la delega. La variabile riservata this viene automaticamente legata all'oggetto ricevente la richiesta di accesso e non cambia; in questo modo l'esecuzione di un metodo di una superclasse avviene come se fosse un metodo dell'oggetto originale. D'altra parte una invocazione ad un qualsiasi altro metodo comporta un cambiamento di this.

Delega

Un concetto viene rappresentato in due oggetti uno con i tratti comuni e l'altro prototipico contenente gli aspetti specifici delle istanze. Gli attributi condivisi risiedono in appositi oggetti (traits). Tali oggetti generalmente non sono concreti nel senso che contengono metodi che accedono a campi non presenti nell'oggetto.

Ogni oggetto mantiene dei riferimenti agli oggetti (traits) con gli attributi condivisi. Quando un oggetto riceve una richiesta di accesso ad un attributo che non possiede la inoltra (delega). La modifica di un attributo condiviso è singola e si propaga automaticamente e inevitabilmente a tutti gli oggetti che condividono l'attributo. L'accesso ad un attributo condiviso avviene tramite delega all'oggetto che lo contiene. La relazione classe/sottoclasse è espressa dalla delega.

Concatenazione

Gli oggetti sono autosufficienti e concreti. Ogni oggetto concettualmente mantiene una propria copia di ogni attributo.

Gli oggetti possono essere definiti senza la necessità di definire una gerarchia di classi. Ogni oggetto che condivide un attributo con altri oggetti ne riceve una copia; il sistema mantiene traccia degli attributi clonati da uno stesso prototipo (clone family). Gli oggetti possono essere modificati come singoli individui. L'accesso all'attributo è diretto, la modifica viene estesa a tutte le repliche. L'accesso ai metodi condivisi è uguale agli altri e non richiedono una diversa interpretazione di self.

Meccanismi di allocazione

Un meccanismo di allocazione consente di ottenere, data una informazione, due informazioni accessibili e modificabili separatamente.

Istanziazione

Definisco singole classi. L'operazione di istanziazione crea un nuovo oggetto a partire dalle definizioni della classe data e di tutte le classi da cui eredita. Gli

attributi non condivisi provengono da tutta la gerarchia di ereditarietà. (copy down). L'istanziazione è un complemento obbligato per i meccanismi di conformità che non definiscono un prototipo delle istanze.

Clonazione

Definisco singoli oggetti. L'operazione di clonazione crea un nuovo oggetto duplicando un prototipo. Un oggetto può assumere il ruolo di prototipo se definisce tutti i campi che usa. I riferimenti al prototipo non vengono coinvolti dalla clonazione.

Oggetto del confronto

Il confronto deve essere fatto alla pari su tre nuclei di linguaggio sostanzialmente identici a meno del meccanismo di condivisione; per questo motivo sono state eliminate oppure ristrette le altre funzionalità. In questo paragrafo vengono elencate le scelte fatte dettate principalmente dalla volontà di rendere minimi i tre linguaggi senza per questo compromettere la generalità del confronto.

Vengono distinti campi e metodi. Sono ammesse solo operazioni sintattiche di aggiunta di campi e di aggiunta e ridefinizione di metodi nell'ambito delle definizioni di nuove classi/oggetti. Sono quindi escluse l'eliminazione di attributi e la possibilità di modificare classi/oggetti già definiti. La condivisione è statica.

Il linguaggio basato su delega supporta in modo naturale i campi condivisi in particolare garantisce la non escludibilità della condivisione. Per avere analoghe garanzie strutturali nei linguaggi basati su ereditarietà e su concatenazione è stata aggiunta una parola chiave per distinguere definizioni di campi condivisi da campi di istanza.

Per uniformare le operazioni di allocazione dinamica è stata fatta la scelta di limitare il dominio di applicabilità della clone nei due linguaggi basati su oggetti. Inoltre, data la non applicabilità del concetto di copia profonda all'istanziazione

di una classe, è stata anche limitata alla superficie del prototipo la profondità della copia (shallow copy). Non sono previsti i costruttori.

Equivalenza dei tre meccanismi di condivisione

In [Sol99a] vengono definite una sintassi e una semantica per i tre linguaggi descritti informalmente nel paragrafo precedente. Le regole semantiche sono esattamente le stesse per tutti i linguaggi a meno di tre funzioni ausiliarie descritte più avanti. Poiché la dimostrazione richiede solo la conoscenza di queste ultime può essere compresa senza una preventiva lettura del testo citato.

Per dimostrare l'equivalenza dei tre linguaggi e quindi dei tre meccanismi di condivisione definisco tre trasformazioni da un linguaggio ad un altro in modo da realizzare una catena chiusa; poi mostro che per un qualsiasi programma si possono eseguire le stesse regole di transizione definite nella semantica. Una trasformazione mappa un albero di entità di un linguaggio in un albero di entità equivalente di un altro linguaggio. L'equivalenza semantica viene dimostrata facendo vedere che le funzioni ausiliarie danno gli stessi risultati quando vengono applicate agli alberi relativi ad una trasformazione.

Per ogni coppia di linguaggi vi sono infinite (banali) trasformazioni che preservano la semantica dei programmi; per individuare una trasformazione si possono uguagliare a due a due le funzioni ausiliarie e ricavare da queste equazioni i vincoli che devono essere soddisfatti dalla trasformazione. La trasformazione esiste se i vincoli sono compatibili. Le trasformazioni definite di seguito vengono ricavate in questo modo pertanto non sono completamente specificate.

Per comodità riepilogo in Tabella 1 le differenze semantiche. I tre linguaggi basati su ereditarietà, delega e concatenazione vengono indicati rispettivamente con $\prec_e$, $\prec_d$, $\prec_c$. La funzione ausiliaria $G(o,a)$ permette di accedere al valore di un campo a di un oggetto o. $G(o,m)$ permette di accedere al corpo di un metodo m di

un oggetto *o*. *U(M,o,a,v)* permette di aggiornare con il valore *v* un campo *a* di un oggetto *o* e restituisce la memoria $\mathcal{M}$ aggiornata.

	Ereditarietà ($\prec_e$)	**Delega ($\prec_d$)**	**Concatenazione ($\prec_c$)**
G(o,a)	Si propaga solo per accedere ai campi condivisi	Si propaga solo per accedere ai campi condivisi	Sempre locale
G(o,m)	Si propaga per accedere ai metodi ereditati	Si propaga per accedere ai metodi non locali (sempre)	Sempre locale (self legato)
U(M,o,a,v)	Singolo aggiornamento; locale per i campi di istanza e remoto per quelli condivisi	Singolo aggiornamento; locale per i campi di istanza e remoto per quelli condivisi	Per i campi di istanza locale e singolo; per i campi condivisi locale più clone family e replicato.
Definizione campi	Sparsi lungo tutta la catena di ereditarietà. I campi condivisi sono marcati	Sparsi lungo la catena di delega: i campi condivisi si trovano solo nei tratti comuni mentre i campi d'istanza si trovano solo nei prototipi	Locali ad ogni prototipo che li contiene. I campi condivisi sono marcati
Definizione metodi	Sparsi lungo tutta la catena di ereditarietà. Possono accedere solo a campi già definiti	Sparsi lungo la catena di delega limitatamente ai tratti. Possono accedere anche ai campi definiti nell'albero di delega sottostante	Locali. Accedono sempre e solo ad attributi locali
New/clone	Crea un oggetto contenente tutti i campi di istanza definiti lungo la catena di ereditarietà.	Copia il prototipo; i delegati non vengono copiati	Copia il prototipo; gli oggetti incorporati vengono implicitamente copiati
oggetto	Limitato ai campi di istanza; i metodi e i campi condivisi sono sparsi nelle classi della catena di ereditarietà	Limitato ai campi di istanza; i metodi e i campi condivisi sono sparsi negli oggetti della catena di delega	Completamente autosufficiente; la clone family permette di risalire a tutte le repliche dei campi condivisi

Tabella 1 Meccanismi di condivisione

Trasformazione da λ_e a λ_d

Per ogni classe E definisco due oggetti D_t e D_p con ruoli rispettivamente di tratto e prototipo. In D_t definisco gli eventuali campi statici di E e tutti i metodi; in D_p definisco tutti i campi di istanza definiti in E e nelle classi che compongono la catena di ereditarietà. Per ogni classe E e superclasse E′: D_p delega a D_t e D_t delega al D_t definito per la superclasse. D_p viene clonato dove E viene istanziato. Se una classe E non viene istanziata non è necessario definire D_p. D_p è clonabile quando E è istanziabile.

Vincolo $G_e(e,a) = G_d(d,a)$

Il dominio di entrambe le funzioni è dato dall'insieme dei campi definiti rispettivamente nella catena di ereditarietà di *e* e nella catena di delega di *d*. Poiché la trasformazione mappa tutti i campi definiti nella catena di ereditarietà sugli oggetti della catena di delega limitandosi a fare degli spostamenti i due domini sono uguali.

Vincolo $G_e(e,m) = G_d(d,m)$

Il dominio di entrambe le funzioni è dato dall'insieme dei metodi definiti rispettivamente nella catena di ereditarietà di *e* e nella catena di delega di *d*. Poiché la trasformazione mappa tutti i metodi definiti nella catena di ereditarietà sugli oggetti della catena di delega limitandosi a fare degli spostamenti i due domini sono uguali.

Vincolo $U_e(M,e,a,v) = U_d(M,d,a,v)$

Il dominio di entrambe le funzioni è dato dall'insieme dei campi definiti rispettivamente nella catena di ereditarietà di *e* e nella catena di delega di *d*. Poiché la trasformazione mappa tutti i campi definiti nella catena di ereditarietà sugli oggetti della catena di delega limitandosi a fare degli spostamenti i due domini sono uguali. In entrambi i casi l'aggiornamento di un campo condiviso avviene in una entità della catena di condivisione comune a tutti gli oggetti che condividono il campo.

Trasformazione da $\lessdot_d$ a $\lessdot_c$

Per ogni prototipo D definisco un oggetto autosufficiente C come segue. Definisco in C tutti gli attributi (campi e metodi) definiti negli oggetti che compongono la catena di delega. I campi definiti nei tratti vengono marcati come campi condivisi in C. Per i metodi ridefiniti includo solo l'ultima definizione. La gerarchia di contenimento riproduce la gerarchia di delega. D e C sono entrambe clonabili oppure non lo sono nessuno dei due.

Vincolo $G_d(d,a) = G_c(c,a)$

Il dominio di G_d su cui la funzione è definita è l'insieme dei campi definiti lungo la catena di delega di *d*; i campi di istanza sono tutti in *d*, i campi condivisi sono sparsi lungo la catena. La trasformazione mappa la definizione di tutti i campi della catena di delega di *d* sull'oggetto autosufficiente *c* limitandosi a marcare i campi condivisi. La funzione G_c cerca i campi localmente pertanto ha lo stesso dominio di G_d.

Vincolo $G_d(d,m) = G_c(c,m)$

Il dominio di G_d su cui la funzione è definita è l'insieme dei metodi definiti lungo la catena di delega di *d*. La trasformazione mappa i metodi definiti lungo la catena di delega di *d* sull'oggetto autosufficiente *c*. La funzione G_c cerca i metodi localmente pertanto ha lo stesso dominio di G_d.

Vincolo $U_d(M,d,a,v) = U_c(M,c,a,v)$

Il dominio di U_d su cui la funzione è definita è l'insieme dei campi definiti lungo la catena di delega di *d*; i campi di istanza sono tutti in *d*, i campi condivisi sono sparsi lungo la catena. La trasformazione mappa la definizione di tutti i campi della catena di delega di *d* sull'oggetto autosufficiente *c* limitandosi a marcare i campi condivisi. La funzione U_c cerca i campi localmente pertanto ha lo stesso dominio di U_c. L'aggiornamento di un campo definito lungo la catena di *d* è osservabile da tutti gli oggetti che condividono quel tratto della catena; analogamente, l'aggiornamento di un campo condiviso in *c* viene replicato in tutti gli oggetti che lo condividono.

Trasformazione da $\lessdot_c$ a $\lessdot_e$

Per ogni clone family di oggetti F(C) definisco una classe E con i soli attributi non incorporati più gli attributi da incorporare. L'attributo *a* di un oggetto C è non incorporato se è definito e usato in C e in nessun altro oggetto C′ della catena di contenimento. Gli attributi da incorporare sono quegli attributi usati ma definiti nel sotto albero di contenimento; se la definizione di un metodo da incorporare è la stessa in tutti i rami del sotto albero di contenimento viene incorporata altrimenti viene incorporato un metodo vuoto; la definizione di un campo da incorporare è la stessa in tutti i rami del sotto albero di contenimento. La gerarchia di contenimento diventa una gerarchia di ereditarietà. La *clone* di un oggetto C viene sostituita dalla *new* della corrispondente classe E; se C è clonabile E è istanziabile.

Vincolo $G_c(c,a) = G_e(e,a)$

Il dominio di G_c su cui la funzione è definita è l'insieme dei campi definiti in *c*. La trasformazione mappa la definizione dei campi di *c* sulla gerarchia di ereditarietà di *e*; ogni campo viene definito in una ed una sola classe. La funzione *new* concentra i campi di istanza in *e*. La funzione G_e cerca i campi di istanza in *e* ed estende la ricerca dei campi condivisi alla gerarchia di ereditarietà pertanto ha lo stesso dominio di G_c. Poiché la gerarchia di ereditarietà è comune a tutte le istanze di una stessa classe i campi definiti lungo la gerarchia sono effettivamente condivisi.

Vincolo $G_c(c,m) = G_e(e,m)$

Il dominio di G_c su cui la funzione è definita è l'insieme dei metodi definiti in *c*. La trasformazione mappa la definizione dei metodi di *c* sulla gerarchia di ereditarietà di *e*; ogni metodo viene definito in una ed una sola classe. La funzione G_e estende la ricerca dei metodi alla gerarchia di ereditarietà pertanto ha lo stesso dominio di G_c.

Vincolo $U_c(M,c,a,v) = U_e(M,e,a,v)$

Il dominio di U_c su cui la funzione è definita è l'insieme dei campi definiti in *c*. La trasformazione mappa la definizione dei campi di *c* sulla gerarchia di ereditarietà di *e*; ogni campo viene definito in una ed una sola classe. La funzione *new* concentra i campi di istanza in *e*. La funzione U_e cerca i campi di istanza in *e* ed estende la ricerca dei campi condivisi alla gerarchia di ereditarietà pertanto ha lo stesso dominio di U_c. L'aggiornamento di un campo condiviso in *c* viene replicato in tutti gli oggetti che lo condividono (clone family); analogamente, poiché la gerarchia di ereditarietà è comune a tutte le sotto classi di una stessa classe, l'aggiornamento di un campo definito in una classe E lungo la gerarchia di *e* è osservabile da tutte le istanze delle sotto classi di E; e poiché la clone family viene mappata in E il campo è osservabile esattamente da tutte le istanze che lo condividono.

Conseguenze

L'equivalenza dei tre meccanismi di condivisione rende intercambiabili in particolare i modelli di implementazione e di rappresentazione.

Modello di implementazione

Il modello di condivisione non impone delle restrizioni al modello di implementazione. Ereditarietà, delega e concatenazione sono tre distinte strategie per implementare un meccanismo di condivisione e possono essere la base per i modelli a classi o a prototipi. Il compilatore può tipicamente scegliere la rappresentazione più appropriata alle altre funzionalità del linguaggio.

Ad esempio nei linguaggi basati su ereditarietà o su delega è opportuno definire a livello di prototipo una tabella dei metodi disponibili in modo da non dovere propagare le richieste alla gerarchia di condivisione. Analogamente in presenza di concatenazione è necessario ricorrere a copia virtuale pena l'impossibilità di avere molti oggetti.

Modello di rappresentazione

Il modello di condivisione non impone vincoli al modello di rappresentazione. Ogni modello ha una rappresentazione che si mappa direttamente sul modello concettuale. Il modello basato su concatenazione richiede un ambiente di programmazione per mostrare e aggiornare gli attributi condivisi perché vengono replicati.

L'esistenza di una gerarchia di ereditarietà o di delega pone il problema di sapere quali attributi possieda un oggetto ovvero a quali messaggi sia in grado di rispondere, in modo, ad esempio, da vedere se gli attributi interagiscono correttamente o è necessario apportare modifiche. Una vista basata su concatenazione mostra tutti gli attributi ed è pertanto la più adatta per questo scopo.

Conclusioni

Ora possiamo chiederci quali siano le differenze intrinseche tra un linguaggio basato su classi e uno basato su oggetti. L'unica differenza è una scelta diversa riguardo a quando determinare le caratteristiche dei prototipi: all'ultimo momento oppure in modo anticipato.

In un linguaggio basato su classi la determinazione degli attributi di un oggetto allocato dinamicamente viene ritardata fino al momento della istanziazione mentre nei linguaggi basati su oggetti questo lavoro viene anticipato alla definizione. I primi, almeno concettualmente, richiedono un operatore di allocazione – new – che ha bisogno di analizzare la gerarchia di ereditarietà come se fosse una ricetta; mentre per i secondi è sufficiente un operatore – clone – che duplica un prototipo precedentemente definito. Nei linguaggi basati su delega il lavoro di definizione del prototipo viene effettuato dal programmatore che deve progettare la gerarchia di delega in modo da separare i prototipi dai tratti comuni. Nei linguaggi basati su concatenazione, è l'ambiente di programmazione che si occupa di eseguire automaticamente il lavoro.

I linguaggi basati su delega mostrano che la distinzione tra sottoclassare e istanziare non è necessaria. Un solo tipo di relazione: delega viene contrapposto ad istanziazione (is a) e sottoclassamento (kind of).

I linguaggi basati su oggetti autosufficienti mostrano che non è necessario definire manualmente le entità non concrete (classi astratte e traits) né preoccuparsi di riorganizzare la gerarchia di condivisione. Questi linguaggi consentono di creare prima concetti individuali e poi di generalizzarli. I sistemi basati su classi o su oggetti con delega richiedono di fornire prima una definizione dell'insieme astratto. Il problema è che è difficile stabilire in anticipo quali siano le caratteristiche essenziali di un concetto.

La differenza tra avere gerarchie di classi o gerarchie di oggetti deleganti da una parte e oggetti autosufficienti dall'altra è ben più rilevante che non la divisione tra basati su classi e basati su oggetti. Nel primo caso il programmatore si deve preoccupare di definire esplicitamente le classi comuni da estendere e gli oggetti condivisi (traits), nel secondo caso il sistema provvede automaticamente a mantenerla e a modificarla.

L'equivalenza dei tre linguaggi ci dice d'altra parte che l'implementazione, la rappresentazione e la tipabilità sono fondamentalmente indipendenti dai tre modelli. Questo ci permette di aggiungere ad un linguaggio esistente le caratteristiche positive degli altri modelli riducendo la scelta del modello ad una questione di gusto e di tradizione. L'equivalenza però significa anche che la scelta del meccanismo di condivisione è ininfluente al fine del raggiungimento dell'Obiettivo di Adattabilità.

In prospettiva i linguaggi basati su concatenazione attaccano la centralità attribuita alla gerarchia di condivisione come elemento organizzante; i linguaggi basati su delega rilassano l'enfasi sull'unità di un oggetto sostituendola con una pluralità raggiungibile dal meccanismo di instradamento.

Capitolo 10

MODELLI BASATI SU PREDICATI

Diversi meccanismi dei linguaggi di programmazione servono per selezionare il metodo più specifico da applicare tra una collezione di candidati.

In questo capitolo descrivo due meccanismi sviluppati entrambi in buona parte da Craig Chambers [Cha93, EKC98]: classi predicato e funzioni con predicato. Le classi predicato si occupano unicamente di modellare gli eventuali aspetti dinamici del comportamento di una classe e lasciano al meccanismo di condivisione il compito di gestire quelli statici. Le funzioni con predicato hanno l'ambizione di modellare sia gli aspetti statici che quelli dinamici del comportamento di una classe e lasciano al meccanismo di condivisione solo la gestione dei campi.

Classi predicato

Le classi predicato sono un costrutto linguistico introdotto da Craig Chambers in [Cha93] che complementa le normali classi. Diversamente da una dasse normale, un oggetto è automaticamente una istanza di una classe predicato ogni volta che soddisfa l'espressione predicato associata alla classe. Per tutto il tempo che un oggetto eredita da una classe predicato, eredita anche tutti i metodi e le variabili di istanza. L'instradamento può dipendere non solo dalla classe dinamica dell'oggetto ma anche dal suo valore o stato.

Una classe predicato ha tutte le caratteristiche di una classe normale, in più ha associato una espressione predicato. Una classe predicato rappresenta il sottoinsieme delle istanze delle sue superclassi che soddisfano l'espressione predicato. Ogni volta che un oggetto è una istanza di una superclasse di una

classe predicato e soddisfa la corrispondente espressione predicato automaticamente eredita anche gli attributi della classe predicato. Una *espressione predicato* è una arbitraria espressione booleana del linguaggio di programmazione sottostante e può contenere attributi delle classi che la classe predicato estende.

In una semplice implementazione delle classi predicato, ogni oggetto viene allocato con anche lo spazio per tutti i campi delle classi predicato che possono specializzarlo e il metodo di instradamento valuta le espressioni predicato per scegliere il destinatario di un messaggio.

Funzioni con predicato

L'instradamento basato su predicati include ed estende le funzionalità dell'instradamento singolo e multiplo, del pattern matching e delle classi predicato. Viene presentato in [EKC98] come una teoria unificata dell'instradamento.

Nell'instradamento basato su predicati, le classi definiscono solo i propri campi; le funzioni vengono definite separatamente. La definizione di una funzione specifica la sua applicabilità con una espressione predicato. Se una funzione deve esibire diversi comportamenti allora si definisce una signature (nome e lista dei tipi dei parametri) comune e si definiscono tante funzioni quanti sono i casi. La selezione di un caso dipende dalla valutazione dei predicati associati. Una funzione è applicabile quando la sua espressione predicato è vera. Una funzione f_1 sostituisce (override) una funzione f_2 quando il predicato di f_1 implica quello di f_2; questa relazione viene calcolata a tempo di compilazione. L'ordine di dichiarazione delle funzioni non è rilevante.

Un predicato è una espressione booleana con tanto di congiunzioni, disgiunzioni e negazioni che viene valutata in un ambiente con i parametri formali legati. Un predicato può contenere test di classi, espressioni sui valori dei campi di una classe e arbitrarie espressioni booleane del linguaggio di programmazione sottostante. Un predicato può avere un nome

Critica ai modelli basati su predicati

I modelli basati su predicati sono dei meccanismi di specializzazione dinamica del comportamento di una classe.

Classi predicato ed ereditarietà dinamica

Self come abbiamo visto è un linguaggio che supporta l'ereditarietà dinamica. Quando un oggetto cambia il delegato sostituisce tutti gli attributi che aveva ereditato dalla gerarchia di delega e mantiene solo la parte incrementale della sua definizione. Con le classi predicato, una classe non cambia mai l'implementazione ereditata, cambia solo una parte del comportamento.

Il meccanismo di condivisione definisce un albero orientato tra le classi; ogni classe è legata alla classe che estende. L'instradamento dei messaggi avviene sempre lungo gli archi di questo grafo cioè nella direzione che va da una classe verso le superclassi. La presenza di un meccanismo di specializzazione dinamica incentrato sugli oggetti permette di cambiare il primo arco dell'albero orientato seguito dalla funzione di instradamento: quello che lega un oggetto alla propria classe.

L'ereditarietà dinamica permette di cambiare la classe di un oggetto senza restrizioni; mentre con le classi predicato ogni oggetto si impegna ad essere almeno di un certo tipo. La differenza è rilevante. La delega dinamica interferisce con il meccanismo di condivisione quindi può causare errori di tipo durante l'esecuzione; le classi predicato no perché, cambiano la classe di un oggetto con delle sottoclassi del tipo atteso dai clienti. In Self dobbiamo definire una gerarchia di delega con il duplice ruolo conflittuale di gerarchia di specializzazione (dinamica) del comportamento e di gerarchia di condivisione (statica) dell'implementazione. Con le classi predicato i due ruoli sono separati: la gerarchia di ereditarietà resta solo un meccanismo di condivisione dell'implementazione mentre le classi predicato specializzano il comportamento.

La capacità di interferire con il meccanismo di condivisione è una esclusiva della delega dinamica ammesso che se ne riesca a trovare un impiego utile; per il resto i due meccanismi aggiungono al modello statico esattamente la stessa flessibilità. Nel capitolo precedente abbiamo constatato che il modello di condivisione non impone delle restrizioni al modello di implementazione. Poiché il meccanismo di condivisione è definito sul meccanismo di instradamento, la delega dinamica lo vincola di fatto alla propria strategia di implementazione che non si distingue per efficienza.

Specializzazione con predicati o con operatore

Le classi predicato sono un meccanismo di specializzazione dinamica del comportamento di una classe. La valutazione dei predicati determina il cambiamento del comportamento che pertanto avviene automaticamente al variare dello stato di un oggetto. È possibile definire un meccanismo di specializzazione dinamica che preveda una operazione esplicita (ad esempio *become*) per cambiare il comportamento di una classe con quello (ri)definito in una sottoclasse. In questo caso, quando si modifica lo stato di un oggetto bisogna anche preoccuparsi di specializzare il comportamento se necessario. I predicati sono espressivi quanto l'operatore esplicito; al più possono richiedere la definizione di funzioni ausiliarie. I predicati hanno il vantaggio di garantire la corretta corrispondenza tra stato e comportamento di un oggetto; mentre l'operatore esplicito permette di scegliere quando (valutare l'espressione per) cambiare stato ed inoltre è più immediato da aggiungere agli attuali linguaggi ad oggetti.

L'operatore esplicito può essere implementato semplicemente sostituendo la tabella di instradamento degli attributi corrente con quella della sottoclasse di specializzazione. L'implementazione delle classi di specializzazione dinamica supera, per un livello di indirezione in meno nell'instradamento, il codice prodotto dall'applicazione del design pattern Strategy in un linguaggio come Java che non supporta la specializzazione dinamica. Le classi predicato possono essere implementate con risultati di poco inferiori. A complicare il problema è la

scelta di quando valutare l'espressione predicato: non appena cambia una variabile (early) oppure quando si accede ai metodi (lazy).

Contributi all'Obiettivo di Adattabilità

I meccanismi di specializzazione dinamica da soli non permettono di definire un'implementazione dell'operazione di differenziazione perché non sono in grado di ridefinire interamente il comportamento di una classe; però permettono di migliorare la soluzione basata su aspetti. Usando le classi di specializzazione dinamica si può eliminare il costo dell'istruzione *if* usata all'inizio di ogni metodo.

Conclusioni

Le classi di specializzazione dinamica sono un meccanismo per cambiare, a tempo di esecuzione, il comportamento di un oggetto in funzione del suo stato. Questo meccanismo ha il vantaggio di rispettare il sistema dei tipi e di fornire una soluzione analoga ma linguistica ai design pattern che si propone di sostituire. Le classi predicato sono un meccanismo molto elegante anche sintatticamente e che offre l'ulteriore garanzia di corrispondenza tra stato e comportamento; essendo però un meccanismo automatico, sottrae al programmatore la scelta di quando valutare i predicati ed è pertanto meno coerente con uno stile di programmazione imperativa.

Capitolo 11

MODELLO AD ATTORI

Il modello ad attori è stato proposto da Carl Hewitt [Hew73] nella tradizione dell'intelligenza artificiale. Il modello è stato sviluppato anche formalmente soprattutto da Gul Agha [AMST93, AghKim98] ma più nella tradizione dei sistemi paralleli e distribuiti. Storicamente, la ricerca sui sistemi paralleli ha privilegiato i problemi di comunicazione intensiva su processori accoppiati strettamente mentre la ricerca sui sistemi distribuiti si è occupata maggiormente di coordinazione, disponibilità, ecc. su processori debolmente accoppiati. Queste diverse assunzioni sull'ambiente di esecuzione dei programmi ha favorito uno sviluppo separato delle due aree di ricerca. Solo di recente si è assistito a un processo di convergenza favorito dallo sviluppo delle tecnologie legate a Internet e di linguaggi neutrali come Java che fanno apparire omogenei tutti i nodi della rete distribuita.

Il modello ad attori viene proposto da Gul Agha [AghKim98] come una estensione alla programmazione orientata agli oggetti mirata alla programmazione su sistemi paralleli e distribuiti. Gli attori rappresentano un modo naturale per integrare la concorrenza con gli oggetti.

Anatomia degli attori

Un attore è un oggetto autonomo: ha uno stato, uno script, una interfaccia e un thread di controllo. Gli attori interagiscono con altri attori scambiando dei messaggi. L'esecuzione del programma segue il flusso dei messaggi. Lo *script* è il metodo che descrive il comportamento dell'attore e che riceve tutti i messaggi. La ricezione di un messaggio determina l'esecuzione atomica del metodo con

parametri specificato. Se il metodo è sconosciuto, lo script può rigettare il messaggio oppure inoltrarlo ad un altro attore a cui viene delegato il compito di eseguirlo. In risposta ad un messaggio, un attore può cambiare stato, spedire messaggi, creare altri attori. Ogni attore ha un riferimento globale unico detto *indirizzo postale* e può spedire messaggi solo agli attori di cui conosce l'indirizzo postale. I messaggi possono essere usati anche per comunicare gli indirizzi postali quindi la topologia di comunicazione degli attori può cambiare dinamicamente. I messaggi sono asincroni, non bloccanti e non ordinati; si possono però definire dei vincoli di sincronizzazione per forzare un ordine nell'esecuzione dei messaggi.

I modelli ad attori forniscono delle primitive di comunicazione, sincronizzazione, gestione dello spazio dei nomi e della memoria.

Critica del modello ad attori

Il modello ad attori non è propriamente né basato su classi né basato su oggetti. Non è neppure necessario implementarlo in un linguaggio orientato agli oggetti; ad esempio in [AMST93] viene descritto un linguaggio funzionale ad attori. Si può interpretare un attore come un oggetto con un metodo – lo *script* – e delle variabili private; oppure come una funzione con delle variabili locali. A far propendere per l'interpretazione ad oggetti è solo il fatto che un attore può creare altri attori.

Non deve stupire pertanto che il modello non possieda un meccanismo di condivisione. Inoltre, dal momento che tutti i messaggi indirizzati ad un attore passano per il metodo di instradamento – lo *script* – non è necessario definire altri metodi in un attore.

I messaggi vengono gestiti in modo atomico e asincrono. Il metodo script viene eseguito in mutua esclusione per garantire l'atomicità. La scelta asincrona permette di esplicitare il parallelismo tra attore mittente e destinatario ma comporta la separazione del messaggio di invocazione da quello di ritorno con

l'eventuale risultato. Il passaggio dei parametri nello stesso messaggio che contiene il metodo è una scelta conservativa non motivata dal modello ma fatta per ricondursi il più possibile vicino ad una definizione classica di funzione.

Gli attori, di base, non supportano nessun particolare modello di esecuzione; inoltre sono un ibrido, non motivato, di un meccanismo di comunicazione fondazionale e di uno *script* ad alto livello. Per usarli, è necessario definire dei protocolli di comunicazione e impegnarsi a rispettarli per ottenere quello che si vuole. È così possibile, ad esempio, definire un meccanismo di condivisione e delle funzioni con valore di ritorno.

Il modello ad attori non prevede una primitiva di comunicazione per porsi in attesa di un particolare messaggio. Pertanto, la definizione di un protocollo di comunicazione che simuli il comportamento di una invocazione sincrona di funzione con valore di ritorno richiede di definire separatamente la continuazione della chiamata a funzione. Inoltre, l'attore che esegue la funzione si deve impegnare ad invocare la sua continuazione sull'attore che ne ha chiesto l'esecuzione passandogli l'eventuale risultato e il contesto della chiamata che gli era stato passato. Questo protocollo non garantisce più l'atomicità di esecuzione di una funzione ma solo delle sottofunzioni (da chiamata a chiamata) in cui costringe a dividere una funzione. D'altra parte l'aggiunta di una primitiva di attesa bloccante avrebbe reso il sistema più fragile.

I thread di esecuzione sono incapsulati negli attori a differenza dei linguaggi concorrenti orientati agli oggetti come Java dove gli oggetti sono separati dai thread. L'incapsulamento dello stato e dei thread di controllo facilita sia la concorrenza che la distribuzione.

Contributi all'Obiettivo di Adattabilità

Facciamo conto di avere integrato il meccanismo di instradamento del modello ad attori in Java nel seguente modo. La classe *Object* definisce un metodo *dispatch* che riceve tutti i messaggi di invocazione di metodo diretti all'oggetto e li

instrada secondo le regole di Java. Tutte le classi, poiché estendono direttamente o indirettamente *Object*, ereditano questo metodo e volendo possono ridefinirlo.

La soluzione proposta consiste nella definizione di una classe *AdaptableObject* che estende (implicitamente) *Object* e aggiunge il metodo di differenziazione con i relativi attributi di servizio e ridefinisce il metodo di instradamento in modo da ridirigere i messaggi sulla copia dell'oggetto selezionata se questo è stato differenziato. Tutte le classi che estendono *AdaptableObject* ereditano il metodo di differenziazione.

Sorgente della soluzione

```
public class AdaptableObject implements Cloneable {
    private Map map;
    private Object selectorParent;
    private Method getSelector;

    public void differentiate(Object selectorParent, Method
getSelector) {
        map = new HashMap();
        this.selectorParent = selectorParent;
        this.getSelector = getSelector;
    }

    private AdaptableObject getTarget() {
        try {
            Object selector =
                    getSelector.invoke(selectorParent, null);
            AdaptableObject target =
                    (AdaptableObject)map.get(selector);
            if (target == null) {
                target = (AdaptableObject)clone();
                target.map = null;
                map.put(selector, target);
            }
            return target;
        } catch (Exception e) {
            System.out.println("getTarget() error"+e);
            return null;
        }
    }

    public dispatch(Method method, Object[] args) {
        if (map != null)
            getTarget().dispatch(method, args);
        else
            super.dispatch(method, args);
    }
}
```

Limiti della soluzione

Questa soluzione è per molti versi simile a quella basata su aspetti. Presenta gli stessi limiti e in più per applicarla bisogna modificare l'intestazione delle classi dei programmi in modo che estendano la classe *AdaptableObject*.

Il limite più grave resta la mancanza di persistenza.

Conclusioni

Con l'estensione di Java proposta è possibile definire una implementazione dell'operazione di differenziazione che ha il merito di funzionare semplicemente estendendo la classe *AdaptableObject*. Purtroppo questa implementazione non produce risultati persistenti e quindi non può essere considerata una soluzione dell'Obiettivo di Adattabilità.

La disponibilità di una funzione di instradamento esplicita e centralizzata unita al meccanismo di condivisione di Java che permette di ridefinirla una volta per tutte, ha reso possibile una implementazione dell'operazione di differenziazione.

In generale però l'utilità di integrare nei linguaggi ad oggetti una funzione di instradamento esplicita appare molto limitata. È preferibile usare la programmazione ad aspetti per raggiungere gli stessi risultati.

Capitolo 12

MECCANISMI DI INSTRADAMENTO

I meccanismi di instradamento si occupano di instradare un messaggio dal mittente al destinatario; cioè a partire dall'invocazione di un metodo o dall'accesso ad una variabile fino alla determinazione del metodo da eseguire o della variabile richiesta.

Sull'instradamento sono basati sia il meccanismo di condivisione (statico) che il meccanismo di specializzazione dinamica.

Il meccanismo di condivisione definisce un grafo orientato tra le classi. Ogni classe è legata alle classi che estende e dalle quali eredita degli attributi. Nei linguaggi con ereditarietà singola (di implementazione) come Java, il grafo si riduce ad un albero con gli archi orientati verso la radice. L'instradamento dei messaggi avviene secondo le regole fissate dal meccanismo di condivisione ma sempre lungo gli archi di questo grafo cioè da una classe verso le superclassi.

Il meccanismo di instradamento può essere incentrato su oggetti oppure su metodi a seconda di dove risieda la funzione di instradamento. Nel primo caso si parla di instradamento singolo, nel secondo di instradamento multiplo.

Per quanto detto nel capitolo sui meccanismi di condivisione, delega e concatenazione sono del tutto equivalenti all'instradamento singolo fornito dai linguaggi basati su ereditarietà pertanto qui possiamo limitarci ad analizzare i meccanismi di instradamento su un modello basato su classi.

Instradamento singolo

I messaggi vengono spediti ad un oggetto ricevente esplicitamente indicato e il suo tipo corrente determina il metodo invocato dal messaggio. Gli eventuali parametri del messaggio vengono passati al metodo invocato ma non partecipano alla sua determinazione.

La funzione di instradamento è incentrata sull'oggetto che riceve il messaggio. I metodi fanno parte degli oggetti; la notazione solitamente usata per inviare messaggi conferma il modello: *oggetto.messaggio(parametri).*

L'instradamento singolo può essere implicito statico, implicito dinamico o esplicito.

Instradamento implicito e statico

La maggior parte dei linguaggi orientati agli oggetti sono basati su instradamento singolo implicito e statico. La funzione di instradamento implicita che viene fornita riceve tutti i messaggi che arrivano ad un oggetto e li instrada fino ai corrispondenti attributi secondo le regole del meccanismo di condivisione. I metodi di una classe possono ridirigere il messaggio ricevuto cioè possono definire esplicitamente una propria funzione di instradamento che estenda quella fornita dal linguaggio.

Avere la parte iniziale dell'instradamento implicita rende più difficile sostituire tutto il comportamento di un oggetto perché se si vuole scrivere una funzione di instradamento esplicito che si sostituisca a quella predefinita è necessario definirla metodo per metodo inoltre non vale automaticamente anche per i metodi ereditati. Per questi è necessario o ridefinirli oppure che siano già stati scritti in accordo al meccanismo di instradamento esplicito che si vuole installare. Inoltre la ridirezione di un messaggio non è una delega e bisogna tener conto della differenza quando si ridirige un messaggio al di fuori dell'oggetto che lo ha ricevuto.

L'operazione di differenziazione richiede di ridefinire interamente il comportamento di un oggetto ma non è possibile farlo in modo generale perché l'instradamento diventa esplicito solo a livello dei singoli metodi definiti nelle varie classi.

Instradamento implicito e dinamico

La funzione di instradamento implicita che viene fornita riceve tutti i messaggi che arrivano ad un oggetto e li instrada fino ai corrispondenti attributi secondo le regole del meccanismo di condivisione. La presenza di un meccanismo di specializzazione dinamica incentrato sugli oggetti permette di cambiare il primo arco del grafo orientato seguito dalla funzione di instradamento: quello che lega un oggetto alla propria classe. Nelle classi predicato, l'instradamento cambia implicitamente al cambiare dello stato dell'oggetto. Nella delega dinamica, l'instradamento cambia in base al valore di una apposita variabile che punta alla classe dell'oggetto. Infine nelle classi di specializzazione dinamica viene fornito un operatore per cambiare la classe di un oggetto con una sua sottoclasse. I messaggi vengono delegati pertanto i metodi definiti nelle classi assegnate dinamicamente possono accedere a tutti i campi privati senza violare l'incapsulamento.

Anche con questo meccanismo, i metodi di una classe possono definire esplicitamente una propria funzione di instradamento ma nella maggior parte dei casi è sufficiente usare il meccanismo di specializzazione dinamica. La possibilità di assegnare una nuova classe ad un oggetto permette di ridefinire anche tutto il comportamento di un oggetto di una certa classe ma non permette di farlo in generale. Di conseguenza non è possibile fornire una soluzione all'Obiettivo di Adattabilità. D'altra parte le classi di specializzazione dinamica permettono di migliorare leggermente la soluzione fornita con la programmazione ad aspetti eliminando la necessità di una condizione all'inizio di ogni metodo.

Instradamento esplicito

L'instradamento esplicito fa parte del modello ad attori. Ogni oggetto definisce esplicitamente la funzione di instradamento responsabile di gestire tutti i messaggi destinati all'oggetto. La funzione può scegliere se gestire localmente un messaggio oppure ridirigerlo; come caso particolare può adottare la politica di instradamento di uno dei meccanismi di condivisione esistenti. Nel modello ad attori i messaggi ridiretti non sono deleghe.

La funzione di instradamento è definita all'interno degli oggetti di conseguenza si deve occupare di gestire tutti i messaggi. Questa particolarità facilita la sostituzione integrale del comportamento di un oggetto e ha permesso di definire una implementazione dell'operazione di differenziazione.

Instradamento multiplo

I tipi degli argomenti di un messaggio possono tutti partecipare alla selezione del metodo da invocare.

La funzione di instradamento è esterna agli oggetti ed è associata al nome del metodo. I metodi non fanno parte di nessuna singola classe e la notazione che meglio rappresenta questa situazione è quella procedurale: *messaggio(parametri)*. È possibile interpretare i multimetodi e i metodi con predicato come se appartenessero simultaneamente a più classi. Un ambiente di programmazione visuale può mostrarli in tutte le classi di appartenenza e volendo si può usare la notazione orientata agli oggetti per i messaggi.

Un oggetto è un aggregato di attributi: campi e metodi. L'instradamento multiplo separa i metodi dagli oggetti e comporta l'adozione di due diversi meccanismi di instradamento: uno tradizionale per i campi e l'altro incentrato sui metodi.

Un oggetto fatto di soli campi ricorda i record dei linguaggi procedurali. Le differenze che permettono di parlare ancora di programmazione ad oggetti sono

l’ereditarietà e l’incapsulamento. I metodi infatti pur essendo esterni alle classi vengono associati con delle regole a una o più classi e possono accedere agli attributi privati solo di queste.

L’instradamento multiplo può essere implicito oppure esplicito.

Instradamento implicito

L’instradamento multiplo implicito viene fornito ad esempio dai multimetodi e dai metodi con predicato.

La funzione di instradamento implicita che viene fornita riceve tutti i messaggi che hanno una certa signature (nome più lista dei parametri) e li instrada fino ai singoli metodi. La funzione di instradamento viene generata automaticamente in base ai parametri formali dei singoli multimetodi oppure in base alle espressioni predicato.

Anche con questo meccanismo, i singoli metodi possono definire esplicitamente una propria funzione di instradamento e ridirigere il messaggio tenendo conto della differenza tra ridirigere e delegare.

L’operazione di differenziazione richiede di ridefinire interamente il comportamento di un oggetto ma non è possibile farlo in modo generale perché l’instradamento multiplo è incentrato sui singoli metodi e non posso conoscere a priori tutte le signature che verranno definite.

Instradamento esplicito

Analogamente a quanto fa il modello ad attori per l’instradamento incentrato sugli oggetti, è possibile definire un instradamento multiplo esplicito. In letteratura vengono descritti solo meccanismi di instradamento multiplo impliciti per i motivi ragionevoli che vedremo.

Per ogni metodo che deve esibire più comportamenti si definiscono tante funzioni quanti sono i diversi comportamenti più una con la stessa signature per l'instradamento.

Una funzione di instradamento esplicita è responsabile di tutti i messaggi compatibili con la propria signature. In base al tipo e allo stato dei parametri attuali deve scegliere la funzione a cui instradare il messaggio.

Per come è definita, una funzione di instradamento esplicita deve conoscere tutte le funzioni che hanno una certa signature. Di conseguenza ogni volta che si vuole aggiungere una sottoclasse che ridefinisca il comportamento di un metodo è necessario modificare manualmente e ricompilare anche la relativa funzione di instradamento. Questa situazione è inaccettabile perché fa perdere uno dei più rilevanti vantaggi della modularità offerta dai linguaggi orientati agli oggetti.

L'instradamento multiplo viene spesso indicato come poco orientato agli oggetti perché come abbiamo visto separa i metodi e il loro instradamento dalle classi. L'idea di separare l'instradamento dei metodi si può applicare più in generale a tutti gli attributi compresi i campi.

Il risultato è un modello disaggregato che pochi sarebbero disposti a considerare orientato agli oggetti. In questo modello, le classi definiscono solo una gerarchia di tipi e un oggetto istanziato rappresenta solo l'identità dell'oggetto ma non contiene gli attributi. Le funzioni di instradamento dei singoli attributi, in base ai parametri attuali devono individuare i corretti destinatari dei messaggi. In particolare le funzioni di instradamento dei campi devono gestire anche la loro allocazione.

L'operazione di differenziazione può essere implementata anche spostando dei campi e modificando di conseguenza dei metodi.

Cambiando la funzione di instradamento di un campo si possono implementare le operazioni sugli attributi come aggiunta e rimozione che sono sufficienti per

spostare un campo da un oggetto ad un altro. Per fare questo cambiamento non serve una funzione di instradamento con un corpo definito esplicitamente ma non basta neppure una funzione implicita costruita in base a dei predicati.

Bisognerebbe trovare un meccanismo di instradamento incentrato sugli attributi, implicito ma componibile esplicitamente nel senso che permettono le espressioni predicato.

Conclusioni

In questo capitolo abbiamo analizzato diverse varianti del meccanismo di instradamento; la Tabella 2 riassume gli aspetti più rilevanti classificando i meccanismi in ordine crescente di flessibilità.

	Tipo	**Proprietà**	**Esempi di applicazione**	**Implementazione operazione di differenziazione**
Più flessibile « Più rigido	Incentrato sull'oggetto *o.metodo(a, b, …)*	Implicito e statico fino al singolo attributo	Java, C++	Nessuna
		Implicito ma dinamico fino al singolo attributo	Classi predicato, delega dinamica	Parziale miglioramento altre soluzioni
		Completamente esplicito	Attori	Soluzione completa
	Incentrato sul metodo *metodo(o, a, b, …)*	Implicito ma dinamico fino al singolo metodo	Metodi con predicato, multimetodi	Nessuna
		Completamente esplicito fino al singolo metodo	Nessuno	Nessuna

Tabella 2 Meccanismi di instradamento

L'operazione di differenziazione può essere implementata solo nel modello ad attori perché è l'unico che permette di ridefinire interamente il comportamento

di un oggetto senza bisogno di conoscere gli attributi che possiede. La soluzione però è inadeguata perché funziona solo a tempo di esecuzione.

Intuitivamente una implementazione accettabile dell'operazione di differenziazione ha a che vedere con il meccanismo di instradamento. Si tratta di spostare dei campi da una classe ad un'altra e fare in modo che i metodi riescano ancora ad accedervi. Per avvicinarci alla soluzione bisognerebbe trovare un meccanismo di instradamento incentrato sugli attributi, implicito ma componibile esplicitamente nel senso permesso dalle espressioni predicato. Purtroppo ragionando dal punto di vista del meccanismo di instradamento sono più i problemi che si aggiungono di quelli che si avviano ad una soluzione.

Nel prossimo capitolo proviamo a cercare questo tipo di soluzione con la riflessività che ha il vantaggio di fornire operazioni analoghe a quelle usate da un programmatore per modificare manualmente il codice sorgente.

Capitolo 13

RIFLESSIVITÀ

Un linguaggio riflessivo orientato agli oggetti permette ad un programma in esecuzione di ispezionare o cambiare gli oggetti che lo costituiscono. Un linguaggio è introspettivo se ha solo la capacità di ispezionare.

L'idea di poter cambiare dinamicamente il codice del programma durante l'esecuzione faceva già parte dell'architettura Von Neumann. Un moderno metalivello riflessivo si distingue per la scelta di fornire strumenti al livello di astrazione del linguaggio anziché a quello dell'eseguibile (linguaggio macchina).

Uno dei principali obiettivi di un linguaggio imperativo e di conseguenza dei linguaggi orientati agli oggetti con un nucleo imperativo è l'efficienza. Una implementazione efficiente di un moderno metalivello riflessivo è particolarmente impegnativa a meno che il linguaggio non preveda già una macchina virtuale con compilatore just-in-time e un codice eseguibile al livello di astrazione del linguaggio. Né il C++ né Java supportano la riflessività; entrambi però dispongono di un metalivello introspettivo che, anche da solo, aggiunge espressività al linguaggio.

La non disponibilità di un metalivello riflessivo nei maggiori linguaggi ad oggetti è motivata, oltre che da difficoltà oggettive, da una precisa scelta. Consentire ad un programmatore di modificare dinamicamente il codice di un programma è considerata potenzialmente una funzionalità troppo pericolosa e per di più non necessaria. Si preferisce cercare di comprendere le esigenze di modifiche dinamiche del comportamento di un programma e modellarle con un meccanismo del linguaggio specifico e strutturato.

Anatomia di un sistema riflessivo orientato agli oggetti

Un sistema riflessivo orientato agli oggetti è composto da operazioni di introspezione e da operazioni di modifica.

Le operazioni di introspezione richiedono che sia definita per ogni entità del linguaggio una corrispondente classe contenente tutte le informazioni specifiche più dei metodi per ottenere liste di altre entità collegate. Ad esempio, da un oggetto libreria si può ottenere la lista delle classi che contiene; da un oggetto classe si può ottenere una lista dei campi o dei metodi; e da un oggetto metodo si può ottenere una versione reificata del suo corpo. Per ottenere un oggetto libreria o classe si può partire dal nome oppure da una istanza. Un metodo reificato è un albero di oggetti che corrisponde grosso modo all'albero di derivazione generato dal compilatore. Per gli oggetti metodo e campo sono definite anche le operazioni per usarli.

Le operazioni di modifica si dividono in operazioni sugli attributi e operazioni sul corpo dei metodi. Le prime permettono di aggiungere/rimuovere un campo o un metodo da una classe. Le seconde permettono di modificare il corpo di un metodo operando sulla sua rappresentazione reificata e di rendere effettive le modifiche. Le operazioni di modifica possono agire a livello di classe o di oggetto; se sono definite a livello di classe possono valere per le sole nuove istanze o per tutte. In un linguaggio basato su classi la scelta più coerente è definire come dominio di azione delle operazioni di modifica tutti gli oggetti compresi quelli già istanziati per motivi di uniformità. In un linguaggio basato su oggetti è invece più coerente far valere le operazioni solo per l'oggetto su cui vengono invocate; se il meccanismo di condivisione usato è la delega, gli effetti delle modifiche influenzano anche tutti gli oggetti deleganti.

La riflessività può essere applicata anche all'implementazione del linguaggio di programmazione. Un linguaggio che supporta la ridefinizione dinamica di parti del sistema è detto *mutabile*. Un linguaggio che supporta l'aggiunta di nuove funzionalità ma non la modifica di quelle predefinite si dice *estendibile*.

Critica all'approccio riflessivo

In un linguaggio di programmazione con un metalivello orientato agli oggetti, la scelta più conveniente è costruire i programmi a partire da oggetti del linguaggio. Il codice eseguibile può essere rappresentato al livello dell'albero di parse anziché come sequenza di byte e nessuna modifica esplicita del codice sorgente è necessaria.

Le operazioni di modifica di un campo, in particolare l'aggiunta, sono costose perché richiedono la rilocazione dell'oggetto e il conseguente aggiornamento di tutti i riferimenti che ha. Una implementazione puntuale richiede di mantenere per ogni oggetto una lista di tutti i riferimenti; oppure si può tenere per ogni classe una lista di tutte le istanze allocate. Una implementazione più rilassata e più efficiente può limitarsi a rilocare l'oggetto lasciando alla vecchia copia il compito di aggiornare i riferimenti in modo lazy. In presenza di un garbage collector, la vecchia copia viene automaticamente eliminata quando tutti i riferimenti sono stati aggiornati.

Le operazioni di modifica del corpo di un metodo sono difficili da implementare solo se l'eseguibile del linguaggio consiste in una sequenza di byte. In questo caso infatti l'operazione di reificazione deve ricostruire una rappresentazione dell'albero di parse a partire dall'eseguibile e l'operazione che applica le modifiche deve usare un compilatore just-in-time.

La macchina virtuale di un linguaggio che definisce come eseguibile una rappresentazione standard del proprio albero di parse può essere veloce almeno quanto il miglior compilatore nativo per quel linguaggio. La dimostrazione è immediata, basta infatti prendere il miglior compilatore nativo per il linguaggio, separare front-end e back-end e distribuire come eseguibile la serializzazione della struttura dati che si passano le due parti. Questa soluzione ha l'ulteriore vantaggio di fornire a costo zero tutte le operazioni di modifica del corpo di un metodo.

Se si adotta come eseguibile una rappresentazione standard dell'albero di parse e si supporta la riflessività allora non è necessario definire un sorgente testuale separato. Un ambiente di programmazione può facilmente ricostruire dinamicamente una rappresentazione testuale del sorgente e usare la riflessività per implementare tutte le operazioni di editing. Tra i vantaggi di questa scelta faccio notare in particolare: la semplificazione (standardizzazione, componibilità) dei tool che operano sui sorgenti e dei preprocessori che aggiungono funzionalità al linguaggio e la possibilità di editare il programma a tempo di esecuzione (ad esempio in fase di debugging).

Contributi all'Obiettivo di Adattabilità

Il problema della ristrutturazione delle classi (*class refactoring*) è stato studiato in particolare da William Opdyke nella sua tesi di dottorato [Opd92]. Per formalizzare il problema si possono identificare due aspetti: primo, la determinazione di un insieme di ristrutturazioni elementari; secondo, la descrizione operazionale di ciò che va fatto per ogni particolare ristrutturazione. In [Opd92] Opdyke propone il seguente insieme di operazioni di ristrutturazione elementari:

- Definire una superclasse astratta per una o più classi esistenti
- Sostituire istruzioni condizionali con classi aggiuntive
- Cambiare una relazione di condivisione (is-a) in composizione (has-a)
- Muovere una classe all'interno e tra gerarchie di classi
- Muovere variabili e metodi tra classi
- Sostituire un segmento di codice con una chiamata a funzione
- Cambiare il nome di una classe, variabile o metodo

- Sostituire un accesso diretto alle variabili con un'interfaccia più astratta

Nessuna di queste operazioni è semplice per quanto alcune possano sembrarlo ad una prima descrizione ad alto livello. Ad esempio, la procedura per muovere attributi (variabili o metodi) richiede l'esecuzione di sette differenti passi alcuni dei quali contengono sottoprocedure e chiamate ad altre procedure di ristrutturazione. Normalmente i programmatori devono eseguire manualmente queste operazioni con conseguenze immaginabili in termini di tempo ed errori.

Opdyke definisce delle precondizioni per applicare queste operazioni al sorgente di un programma che garantiscono la preservazione del comportamento. Per raggiungere l'Obiettivo di Adattabilità noi ci proponiamo di definire con una sequenza di operazioni riflessive una forma di ristrutturazione – la differenziazione – che deve operare anche a tempo di esecuzione e che deve preservare il comportamento del programma a meno di una modifica circoscritta e ben definita.

L'implementazione dell'operazione di differenziazione usa la procedura per sostituire un accesso diretto alle variabili con un'interfaccia più astratta e quella per muovere variabili e metodi. Purtroppo quest'ultima, per garantire la preservazione del comportamento, è definita da Opdyke o tra classi della stessa gerarchia di ereditarietà oppure tra la classe di un composito e le classi dei componenti a patto però che gli oggetti componenti appartengano in modo esclusivo ad un solo oggetto composito. Nel nostro caso sono due classi qualsiasi ma soprattutto, trattandosi di una operazione a tempo di esecuzione non si tratta solo di spostare un attributo da una classe ad un'altra ma di spostarlo da un oggetto ad una molteplicità di altri oggetti.

Descrizione operazionale della soluzione

Chiamo *obj* e *objType* rispettivamente l'oggetto da differenziare e il suo tipo. Chiamo *clientType* e *objRef* rispettivamente i tipi degli oggetti cliente e la variabile che contiene un riferimento a *obj.* Chiamo *targetType* il tipo degli oggetti

destinatari delle copie differenziate di *obj*. Infine chiamo *selectorMethod* e *selectorObj* rispettivamente il metodo che ritorna il destinatario corrente e l'oggetto che contiene il metodo. La soluzione descritta fa uso di sottoprocedure definite in [Opd92] per evitare di complicare l'algoritmo con dettagli minori.

1. Aggiungo alla classe *targetType* e a tutte le sue istanze una variabile di tipo *objType* e i due metodi per accedervi. Uso le procedure: *create_member_variable* e *create_member_function*. Inizializzo le variabili con copie distinte di *obj*.

2. Aggiungo alle classi *clientType* e a tutte le loro istanze la variabile *selectorObj* inizializzata con il parametro dell'operazione di differenziazione nelle istanze che contengono un riferimento a *obj*. Uso la procedura: *create_member_variable*.

3. Astraggo nelle classi *clientType* e in tutte le loro istanze l'accesso alla variabile *objRef*. Uso la procedura: *abstract_access_to_member_variable*. Reinizializzo *objRef* a null nelle istanze che referenziano *obj* il quale di conseguenza viene eliminato.

4. Aggiungo nelle classi *clientType* e in tutte le loro istanze, all'inizio del metodo *getObjRef* eventualmente aggiunto al passo precedente, un'istruzione condizionale che, se *objRef* è nulla, restituisce la copia di *obj* contenuta nel *targetObj* fornito dal *selectorMethod* applicato a *selectorObj*. Uso delle operazioni riflessive sulla rappresentazione reificata del metodo.

Limiti della soluzione

La soluzione modifica la dimensione degli oggetti quindi può interferire, tra gli altri, con il meccanismo di serializzazione che senza ulteriori correzioni può perdere la compatibilità all'indietro. Le soluzioni basate su aspetti e su attori creano le copie di *obj* in modo lazy; la stessa ottimizzazione può essere aggiunta alla soluzione riflessiva.

Rispetto alle precedenti soluzioni che non facevano uso della riflessività questa implementazione assicura un tempo di esecuzione molto peggiore per

l'operazione di differenziazione (lineare sul numero di clienti e destinatari potenziali esistenti contro una costante bassissima); in seguito le operazioni sulle copie di *obj* eseguite dai clienti sono più efficienti di un fattore costante (un livello di indirezione in meno e un accesso in meno ad una mappa). Per quanto riguarda la complessità misurata in spazio occupato, la riflessività richiede per ogni oggetto cliente o destinatario uno spazio aggiuntivo pari a un riferimento ma solo se viene usata l'operazione di differenziazione. Invece le altre soluzioni si riservano comunque e in ogni oggetto lo spazio per tre riferimenti in più, se viene usata la differenziazione, allocano una quantità di spazio proporzionale al numero dei destinatari.

Anche questa soluzione funziona solo a tempo di esecuzione in quanto non si preoccupa di trovare o definire dei percorsi per inizializzare le nuove variabili aggiunte a *targetType* e ai *clientType*. Per farla diventare una ristrutturazione permanente del programma bisogna risolvere anche i seguenti problemi:

- Modifico *targetType* oppure creo una sottoclasse? Se tutte le istanze di *targetType* possono essere restituite da *methodSelector* e non si prevede di usarlo in altro modo allora è ragionevole modificarlo direttamente; in alternativa forse è meglio definire una sottoclasse.

- Chi inizializza la copia di *obj* in *targetType*? Posso crearla nel costruttore e inizializzarla con valori di default oppure posso aggiungere un parametro ai costruttori e farmela passare o ancora posso lasciarla non inizializzata e fare in modo che chi la usa invochi prima il metodo di aggiornamento. Le ultime due ipotesi spostano ricorsivamente il problema sulle classi che usano *targetType*.

- Come inizializzo *selectorObj* nelle classi *clientType*? Può non servire nemmeno una ulteriore variabile *selectorObj* perché magari ho già una variabile assegnata allo stesso oggetto oppure posso arrivarci indirettamente tramite un oggetto

che ho già. Oppure serve e posso farmela passare nel costruttore ma allora devo pormi ricorsivamente il problema nelle classi che usano *clientType*.

- Posso eliminare la variabile *objVar* nelle classi *clientType* e semplificare la soluzione? Se viene assegnata solo nel costruttore probabilmente si; altrimenti devo cercare tutti i metodi di tutte le classi che aggiornano la variabile *objVar* e devo stabilire se si tratta solo di una inizializzazione o comunque di una scelta progettuale legata a quella di non differenziare *objVar* e quindi posso eliminarla oppure se serve mantenerla.

Tutte queste domande hanno in comune una cosa: sono problemi di progettazione e richiedono inventiva. In genere ammettono più soluzioni molto diverse tra loro che corrispondono ad altrettante scelte progettuali dipendenti dal dominio del problema. Anche concedendo la possibilità di inventare una soluzione funzionante facendo una analisi statica del programma, molto difficilmente tale soluzione potrà essere riconosciuta come propria dai programmatori che hanno scritto il sorgente iniziale. Bisognerebbe perlomeno coinvolgerli nelle scelte interagendo con loro e comunque si porrebbe il problema del formato dei file dei documenti e il problema della proliferazione di versioni incompatibili e non aggiornabili del programma.

Conclusioni

La riflessività, di suo, non è la soluzione di nessun problema; è uno strumento sufficiente per implementare nuove funzionalità su un linguaggio esistente. In genere, quelle stesse funzionalità possono essere implementate nel compilatore o nella macchina virtuale senza ricorrere alla riflessività che pertanto si rivela uno strumento non necessario ed eccessivamente potente. I vantaggi della riflessività non vanno quindi ricercati in ciò che permette di fare ma nella possibilità che offre alla comunità dei programmatori di fare evolvere il linguaggio e gli strumenti collegati autonomamente e in concorrenza sulle funzionalità aggiunte.

La riflessività consente di superare il dualismo sorgente/eseguibile favorendo enormemente lo sviluppo e l'integrazione di tutti gli strumenti coinvolti nel processo di produzione del software.

Le operazioni di ristrutturazione elementari definite in [Opd92] possono essere implementate più facilmente se il sorgente è in una *forma riflessiva* anziché testuale. Affinché questi strumenti di automazione vengano pienamente accettati dai programmatori è necessario eseguire le operazioni in modo interattivo per far riconoscere come proprio il risultato della ristrutturazione.

Per quanto detto sugli aspetti implementativi, si raccomanda anche per Java la sostituzione del sorgente testuale e degli eseguibili in *bytecode* con una rappresentazione definita al livello di astrazione dell'albero di derivazione e un pieno supporto alla riflessività (ora in Java si può parlare quasi esclusivamente di introspezione). Questo permetterebbe anche una semplificazione del compilatore adattivo presente nella macchina virtuale.

La riflessività consente di implementare l'operazione di differenziazione migliorando da alcuni punti di vista le altre implementazioni definite; purtroppo resta il problema della persistenza.

La riflessività mette a disposizione delle operazioni sufficienti, nelle mani di un programmatore, a realizzare la soluzione che preferisce. Purtroppo non è possibile raggiungere lo stesso risultato automaticamente perché la soluzione richiede scelte progettuali legate anche al dominio del problema che devono essere condivise dal programmatore del sorgente iniziale che è l'unico garante della linearità dello sviluppo del programma e della compatibilità dei formati dei file.

Capitolo 14

CONCLUSIONI

Nella prima parte della tesi, la piattaforma Java è stata riconosciuta come punto di riferimento per i linguaggi di programmazione orientata agli oggetti. In questa, ho mostrato che non è possibile raggiungere l'Obiettivo di Adattabilità restando nell'ambito di questo paradigma. A conclusione di questa parte mi propongo di fare due cose: una è promuovere un determinato sviluppo degli attuali linguaggi ad oggetti in particolare Java; l'altra è cercare di formulare domande significative per comprendere da dove ricominciare la ricerca di una soluzione.

Sull'evoluzione dei linguaggi orientati agli oggetti

Limitatamente agli aspetti dei linguaggi di programmazione orientata agli oggetti che sono stati analizzati in questa parte, sono emersi motivi per sostenere l'opportunità di integrare alcune nuove funzionalità nei linguaggi attuali. L'analisi ci ha inoltre consentito di trovare argomenti per preferire nettamente alcuni meccanismi rispetto ad altri per implementare queste funzionalità.

La programmazione basata su oggetti contrappone un diverso modello concettuale a quello della programmazione basata su classi. A parte una valutazione di maggior concretezza e semplicità che può essere più o meno condivisa, abbiamo visto che è del tutto equivalente al modello basato su classi. In particolare l'implementazione, la rappresentazione e la tipabilità sono fondamentalmente indipendenti dai due modelli. L'unica differenza è una scelta diversa riguardo a quando determinare le caratteristiche dei prototipi: rispettivamente in modo anticipato oppure all'ultimo momento. Quindi il preferire l'uno o l'altro è una questione di gusto e tradizione e dal momento che

i due linguaggi di programmazione ad oggetti più diffusi – Java e C++ – sono entrambi basati su classi la scelta è quasi obbligata.

Il primo obiettivo è supportare pienamente i design pattern a livello di linguaggio di programmazione. I design pattern hanno già dimostrato di poter contribuire in modo rilevante ad aumentare la flessibilità e la riusabilità del software. A livello di linguaggio di programmazione si può migliorare la località di espressione usando gli aspetti e si può migliorare l'efficienza dei design pattern dinamici (State, Strategy, …) usando le classi di specializzazione dinamica.

Gli aspetti permettono di concentrare in un unico modulo tutto il codice e i dati di una funzionalità del programma che altrimenti verrebbe dispersa tra le classi. In fase di precompilazione, tutti gli attributi definiti in un aspetto vengono distribuiti nuovamente nel sorgente del programma; ognuno viene aggiunto nei punti designati. Gli aspetti aumentano la località di espressione e riducono la quantità di codice da scrivere. Gli aspetti vanno però intesi unicamente come manipolatori statici del sorgente nel senso che ho appena detto. L'attuale implementazione di AspectJ [Xer99] ha ancora una capacità limitata di designare punti del sorgente dove fare le aggiunte e d'altra parte offre anche una interpretazione dinamica degli aspetti che usa un meccanismo non competitivo rispetto a quello descritto qui di seguito.

Le classi di specializzazione dinamica sono un meccanismo per cambiare a tempo di esecuzione il comportamento di un oggetto in funzione del suo stato. Per ogni classe si possono definire delle sottoclassi di specializzazione dinamica, ciascuna ridefinisce una parte del comportamento della classe. Usando esplicitamente l'operatore predefinito *become* un oggetto può assumere il comportamento definito in una sua sottoclasse di specializzazione. Questo meccanismo ha il vantaggio di rispettare il sistema dei tipi e di fornire una soluzione analoga ma linguistica ai design pattern che si propone di sostituire. Le classi predicato [Cha93] sono un meccanismo molto elegante anche

sintatticamente e che offre l'ulteriore garanzia di corrispondenza tra stato e comportamento; essendo però un meccanismo automatico, sottrae al programmatore la scelta di quando valutare i predicati ed è pertanto meno coerente con uno stile di programmazione imperativa.

Ho già avuto modo di osservare che uno dei maggiori punti di forza di Java è la presenza di una macchina virtuale. Il codice eseguibile dovrebbe però essere definito al livello dell'albero di parse anziché come sequenza di byte e, facilitati anche da questo cambiamento, si dovrebbe supportare pienamente la riflessività.

La riflessività rende non necessaria la definizione di un sorgente testuale separato e favorisce enormemente lo sviluppo e l'integrazione di tutti gli strumenti coinvolti nel processo di produzione del software. Inoltre offre alla comunità dei programmatori la possibilità di fare evolvere il linguaggio e gli strumenti collegati autonomamente e in concorrenza sulle funzionalità aggiunte da ciascuno.

Sul raggiungimento dell'Obiettivo di Adattabilità

Tutti i modi che ho trovato per implementare l'operazione di differenziazione richiedono di ridefinire interamente il comportamento degli oggetti che si vuole differenziare.

Con gli aspetti si può implementare l'operazione di differenziazione perché, tramite i designatori, consentono di aggiungere il comportamento differenziato in tutti i metodi di tutte le classi. Per applicare la soluzione è sufficiente ricompilare i programmi.

Con gli attori si può implementare l'operazione di differenziazione perché il metodo di instradamento esplicito può contenere il comportamento differenziato per tutti i metodi di una classe. Per applicare la soluzione è necessario modificare l'intestazione delle classi in modo che estendano la classe che definisce il comportamento differenziato.

Con la riflessività si può implementare l'operazione di differenziazione perché fornisce operazioni per spostare campi tra oggetti e per aggiungere a tutti i metodi che accedono ad un oggetto il comportamento differenziato. Per applicare la soluzione non è necessario ricompilare i programmi.

Purtroppo queste implementazioni non producono risultati persistenti e quindi non possono essere considerate soluzioni dell'Obiettivo di Adattabilità. Durante l'esecuzione del programma posso applicare l'operazione di differenziazione ma al termine del programma perdo tutte le modifiche fatte. Un programmatore che decidesse di supportare le funzionalità aggiunte eseguendo una sequenza di operazioni di differenziazione si troverebbe a dover fare un grosso lavoro manuale di riprogettazione e implementazione. Inoltre il risultato della applicazione dell'operazione di differenziazione non assomiglia minimamente alla soluzione che può essere scritta da un programmatore.

La riflessività mette a disposizione delle operazioni sufficienti, nelle mani di un programmatore, a realizzare la soluzione che preferisce. Purtroppo non è possibile raggiungere lo stesso risultato automaticamente perché esistono più soluzioni molto diverse tra loro che corrispondono ad altrettante scelte progettuali dipendenti dal dominio del problema. Anche concedendo la possibilità di generare automaticamente una soluzione funzionante, molto difficilmente tale soluzione può essere riconosciuta come propria dai programmatori che hanno scritto il codice sorgente iniziale. Bisognerebbe perlomeno coinvolgerli nelle scelte interagendo con loro e comunque si porrebbe il problema del formato dei file dei documenti che usano funzionalità aggiuntive e il problema della proliferazione di versioni incompatibili del programma e non aggiornabili senza perdere le nuove funzionalità.

I programmatori di una applicazione sono gli unici garanti della linearità dello sviluppo di nuove versioni e della compatibilità dei formati dei file.

Riassumendo, da una parte abbiamo trovato che non è possibile implementare l'operazione di differenziazione come ristrutturazione automatica e persistente

delle classi. D'altra parte, la possibilità di implementare l'operazione di differenziazione seppure a tempo di esecuzione dimostra che una singola operazione può tradurre una scelta progettuale che solitamente viene dispersa nella struttura del programma.

Questo fatto suggerisce l'idea che la programmazione ad oggetti obbliga il programmatore a sovraspecificare la struttura dei programmi e che la ricerca di una ristrutturazione delle classi simile a quella che progetterebbe un programmatore sia necessaria nel paradigma di programmazione ad oggetti ma non in generale. L'operazione di differenziazione dà l'intuizione che si può definire un nuovo paradigma di programmazione che non richiede al programmatore di assumersi la responsabilità di definire certi aspetti della struttura del programma.

Proviamo dunque ad esplorare l'ipotesi che si può fare a meno sia di definire i percorsi di inizializzazione per i dati sia di definire la composizione delle classi. Finora abbiamo usato la ristrutturazione manuale delle classi come punto di riferimento e abbiamo valutato le implementazioni dell'operazione di differenziazione trovate rapportandole ad essa. Ora ci proponiamo di rivalutare le implementazioni trovate per orientarci verso una soluzione corretta.

Le implementazioni dell'operazione di differenziazione tendono ad organizzare gli oggetti in modo omogeneo: le istanze di una classe sono tenute raggruppate e nascoste. Solo un numero ristretto di istanze ha bisogno di essere esposto ai clienti della classe. I clienti non hanno cioè la necessità di precisare esattamente le istanze che usano.

L'operazione di differenziazione aggiunge implicitamente un parametro a tutti i metodi di un oggetto differenziato. Inoltre fornisce un meccanismo per reperire implicitamente il parametro aggiunto.

Tutti le implementazioni dell'operazione di differenziazione richiedono di ridefinire interamente il comportamento degli oggetti che si vuole differenziare perché è l'unico modo per aggirare la rigidità di un puntatore.

Una variabile puntatore sia che si tratti di un campo di un oggetto o di un parametro di un metodo, viene assegnata univocamente ad un oggetto. Nel senso che ogni invocazione di metodo o accesso a campo passa per un puntatore che determina su quale oggetto deve essere eseguita. Ridefinendo tutto il comportamento di un oggetto, la dereferenziazione di un puntatore cliente viene (nuovamente) parametrizzata dalla variabile di selezione del destinatario e restituisce un oggetto dello stesso tipo ma diverso da quello legato al puntatore.

Un percorso di inizializzazione di un oggetto è un albero di chiamate annidate a funzioni che lo richiedono come parametro attuale: a partire dalla funzione che alloca l'oggetto fino alla prima di ogni ramo che lo usa o lo assegna ad un campo del proprio oggetto per usarlo in seguito. Tutte le funzioni interne a questo albero (escluse quindi radice e foglie) si limitano a passare il riferimento, cioè non sono affatto interessate al particolare oggetto puntato, lo passano solo perché serve ad una funzione da loro direttamente o indirettamente chiamata.

Devo trovare un meccanismo che sostituisca i puntatori lasciando aperta la determinazione dell'oggetto puntato. Nella definizione di una funzione, quando invoco un'altra funzione lo faccio perché sono interessato al risultato che produce; in generale il mio algoritmo non richiede di vincolare tutti i parametri attuali della chiamata. Anche quando ho bisogno di passare un riferimento ad un oggetto, in genere, mi basta fissare il vincolo che la funzione chiamata usi lo stesso oggetto che uso io non ho bisogno di precisare esattamente quale.

Non è un problema di cambiare un puntatore in modo da farlo puntare ad un altro oggetto. Se il chiamante passa un oggetto e il chiamato ne usa un altro è inutile il parametro. Il problema che si pone è trovare una alternativa ai puntatori che permetta di legare chiamante e chiamato ad un oggetto senza

affidare la responsabilità dell'inizializzazione del puntatore al (percorso) chiamante.

I linguaggi di programmazione orientati agli oggetti obbligano il programmatore a sovraspecificare i programmi. I metodi fanno affidamento su una struttura statica degli oggetti e richiedono sempre di specificare l'instradamento dei dati, al massimo permettono di ripartire la responsabilità tra più classi (aumentando però i vincoli di dipendenza).

I linguaggi basati su oggetti autosufficienti [Tai92] mostrano che non è necessario definire manualmente le classi astratte né preoccuparsi di (ri)organizzare la gerarchia di ereditarietà, il sistema può provvedere automaticamente a mantenerla e a modificarla.

Le funzioni con predicato [EKC98] mostrano che l'instradamento di un'invocazione a metodo può essere incentrato sul metodo anziché su un oggetto.

P A R T E I I I

ALTERNATIVE FONDAZIONALI AL PARADIGMA OO

Nella parte precedente ho mostrato che il paradigma di programmazione orientato agli oggetti è inadeguato per raggiungere l'Obiettivo di Adattabilità e che non è possibile modificarlo utilmente restando all'interno del paradigma. Ci si può legittimamente chiedere se, facendo un passo indietro, il paradigma imperativo sottostante ovvero altri paradigmi alternativi – funzionale, dataflow, logico - siano adeguati e possano essere presi come base per un nuovo orientamento.

In questa parte analizzo i quattro paradigmi di programmazione: imperativo, funzionale, dataflow e logico. Di ognuno, partendo dalle tradizioni in cui è nato, presento il modello computazionale e analizzo i concetti più rilevanti che sono stati sviluppati nell'ambito di quel paradigma. Il *modello computazionale* descrive come un programma deve essere valutato. La parola *tradizione* è usata qui nel senso più ampio di pre-comprensione per mettere in evidenza la storicità del contesto in cui viene fatta ricerca. Tutte le domande nascono da una tradizione che orienta la ricerca e apre lo spazio delle possibili risposte. Il problema non è cercare le risposte giuste ma formulare domande significative. Nella parte di analisi critica di ciascun paradigma elenco le scelte arbitrarie e le sovraspecificazioni obbligate mettendo in evidenza le risposte che non possono essere date e le domande che non possono essere neppure formulate restando in quel paradigma. Ad esempio, nei paradigmi imperativo e funzionale, porsi il problema di definire in una chiamata a funzione più parametri attuali per uno stesso parametro formale è incomprensibile perché prevale l'idea che i parametri debbano essere passati dal chiamante; mentre nel paradigma logico è normale definire più regole per raggiungere uno stesso obiettivo ed aspettarsi che sia il motore inferenziale a sceglierne una.

La ricerca di una soluzione all'Obiettivo di Adattabilità prosegue cercando di dare una risposta alla seguente domanda sintetizzata generalizzando le conclusioni della parte precedente.

Posso ripartire la responsabilità della determinazione dei parametri attuali tra la chiamata a funzione e la funzione chiamata?

Più esplicitamente il problema si articola nelle seguenti tre domande. Posso scrivere una funzione che possa essere chiamata con un sottoinsieme (eventualmente vuoto) dei parametri di cui ha bisogno per produrre il risultato e che provveda a farsi calcolare quelli mancanti? Posso estendere esplicitamente e dinamicamente l'algoritmo di ricerca/produzione dei parametri attuali? E posso fare in modo che i parametri attuali determinati separatamente in questo modo formino un unico contesto chiamante; in particolare non interferiscano con altre chiamate?

La soluzione associata ad una eventuale risposta affermativa risolve anche come casi particolari i seguenti problemi. Posso trovare un sostituto ai puntatori che lasci aperta la determinazione dell'oggetto puntato? Posto che un oggetto è un aggregato di attributi e che pertanto un puntatore mi permette di raggiungere tutti gli attributi dell'oggetto puntato, posso rappresentare un oggetto in forma disaggregata ed usare il sostituto del puntatore per raggiungere ugualmente tutti gli attributi di un oggetto come se fosse ancora unitario? E posso come programmatore sottrarmi alla responsabilità di definire la struttura per rappresentare una entità complessa?

Capitolo 15

PROGRAMMAZIONE IMPERATIVA

La programmazione imperativa nasce dalla tradizione dell'architettura delle macchine di von Neumann. Le scelte progettuali alla base dei linguaggi imperativi sono dominate dalla preoccupazione per l'efficienza che si manifesta come stretta relazione tra i linguaggi e l'architettura su cui vengono eseguiti.

In particolare il modello di memoria con un indirizzamento lineare e una grande quantità di celle che contengono dati modificabili, ha influenzato il concetto di variabile e spiega la grande importanza dell'istruzione di assegnamento nei linguaggi imperativi mentre la presenza di una singola unità di calcolo e del programma memorizzato nella stessa memoria sequenziale dei dati ha privilegiato l'esecuzione sequenziale e l'iterazione.

L'esecuzione sequenziale è una delle caratteristiche essenziali dell'architettura von Neumann. I linguaggi di programmazione imperativi costringono il programmatore a scrivere i programmi in modo sequenziale. L'eventuale supporto alla concorrenza che viene fornito è adatto unicamente per specificare concorrenza tra processi, non certo per esplicitare il parallelismo di un'espressione.

L'architettura von Neumann

L'architettura di computer proposta da von Neumann nel 1945 comprende le seguenti tre unità collegate da un bus: dispositivi di ingresso/uscita, memoria e CPU. I dispositivi di ingresso/uscita trasferiscono dati da e verso l'ambiente esterno cioè permettono di interagire con la macchina. La memoria contiene sia il programma da eseguire che i dati su cui opera il programma. La CPU è

responsabile dell'esecuzione del programma; legge il programma dalla memoria sequenzialmente una istruzione per volta, la esegue e fa qualcosa con il risultato.

I moderni personal computer hanno un'architettura più evoluta ma sempre riconducibile a quella di Von Neumann; usano nuovi tipi di dispositivi di ingresso/uscita molta più memoria primaria e secondaria e un processore (raramente due) molto più veloce. Anche il solo aumento quantitativo delle risorse di calcolo e memoria rende possibili applicazioni che in precedenza non lo erano.

Il modello di computazione imperativo

I linguaggi imperativi si basano sul modello di computazione control-flow proposto da von Neumann. Questo modello definisce un programma una sequenza di istruzioni indirizzabili, ognuna delle quali specifica una operazione e le locazioni di memoria degli operandi oppure specifica il trasferimento di controllo ad un'altra istruzione incondizionatamente o al verificarsi di una condizione. Il metodo di esecuzione consiste nel partire dalla prima istruzione di un programma, eseguirla, e procedere con la successiva a meno che l'istruzione eseguita non richieda un trasferimento di controllo. In quest'ultimo caso, l'esecuzione continua dall'istruzione che si trova dove è stato trasferito il controllo. Un modello di computazione basato sul flusso di controllo specifica l'istruzione successiva da eseguire in base all'esito dell'esecuzione dell'istruzione corrente. Anche nelle varianti parallele del modello von Neumann, la sequenza delle istruzioni è controllata esplicitamente dal programmatore o dal compilatore.

Anatomia dei linguaggi imperativi

Un programma in un linguaggio imperativo è organizzato in funzioni e dati. Nella definizione delle funzioni, il meccanismo privilegiato per ripetere una sequenza di istruzioni è l'iterazione.

Funzioni

Una funzione è una sequenza di istruzioni con un inizio ed una fine propri; richiede in ingresso un elenco di parametri (formali) ed eventualmente produce in uscita un valore. Una chiamata a funzione trasferisce il controllo all'inizio della funzione e lo riprende alla fine; passa tutti i parametri (attuali) richiesti contestualmente alla chiamata e al ritorno riceve l'eventuale risultato.

Variabili e assegnamento

Una variabile è un identificatore legato ad un valore che può essere cambiato da una istruzione di assegnamento durante l'esecuzione del programma.

Le variabili e l'assegnamento sono necessari in un linguaggio imperativo per diversi motivi: per mantenere il risultato del calcolo di una sottoespressione; assieme ad una istruzione condizionale; per marcare il progresso di una iterazione; per costruire strutture dati complesse.

Un programma è una sequenza di istruzioni, ogni valore calcolato deve essere memorizzato cioè assegnato ad una cella di memoria per poter essere riusato da un'altra istruzione più avanti nella sequenza altrimenti deve essere ricalcolato cosa non sempre possibile e comunque inutilmente costosa.

Se il modo di calcolare un valore intermedio dipende da una condizione allora è necessario usare una istruzione condizionale che in ogni ramo assegni ad una apposita variabile il valore opportunamente calcolato. Se il linguaggio ammette più punti di ritorno per una funzione è possibile anche se non conveniente definire una funzione contenente l'istruzione condizionale che in ogni ramo ritorni il valore opportunamente calcolato.

Un ciclo richiede almeno una variabile che ad ogni iterazione possa essere assegnata ad un nuovo valore finché non viene soddisfatta la condizione di fine ciclo. I cicli *for* definiscono esplicitamente una variabile locale che cambia valore ad ogni iterazione; i cicli *while* e *repeat* valutano ad ogni iterazione una espressione che per poter cambiare valore deve contenere direttamente o indirettamente

almeno una variabile che venga riassegnata nel corpo del ciclo o nell'espressione stessa. Fanno naturalmente eccezione i cicli infiniti che non necessitano di variabili.

Le istruzioni operano su dati e da sole possono esprimere solo la parte algoritmica di un programma; le entità (documenti, preferenze, interfaccia utente, …) devono essere modellate con apposite strutture dati mantenute in memoria. I dati possono essere raggruppati in modo omogeneo come vettori/matrici o in modo non omogeneo come strutture (dette anche record o aggregati).

Iterazione

I programmi sono di lunghezza finita; le istruzioni che li compongono sono conservate nella memoria. L'unico modo per realizzare calcoli complessi è la ripetizione di una sequenza di istruzioni. Un programma in un linguaggio imperativo di solito fa uso di iterazione anziché ricorsione perché è il modo più efficiente per ripetere una sequenza in una macchina von Neumann. La ricorsione infatti richiede (concettualmente) una quantità variabile di spazio per lo stack e più tempo per la chiamata e il ritorno da funzione.

Critica del modello imperativo

I linguaggi imperativi sono troppo legati all'architettura von Neumann. L'errore sta nel pensare che per essere efficienti su una macchina (con architettura von Neumann) bisogna fornire al linguaggio un modello di esecuzione e delle astrazioni mappabili direttamente su questa macchina cioè che esista un trade-off tra astrazioni ed efficienza.

Il linguaggio Java ha contribuito con decisione ad affermare l'idea che tra un linguaggio di programmazione ed una macchina concreta debba essere frapposta una macchina virtuale con due scopi: neutralità e supporto al livello di astrazione del linguaggio. La neutralità rispetto alle singole piattaforme hardware rende i programmi Java eseguibili ovunque sia disponibile una JVM. Il supporto alle

astrazioni del linguaggio come creazione e manipolazione di oggetti, invocazione di metodi con instradamento e gestione delle eccezioni facilita la compilazione dei programmi. La macchina virtuale Java ha centrato entrambi gli obiettivi. Purtroppo però essendo il nucleo del linguaggio Java imperativo non è possibile apprezzare i vantaggi derivanti da una macchina virtuale con un modello di esecuzione al livello di astrazione del linguaggio ma diverso da quello di von Neumann. Inoltre le implementazioni attuali della macchina virtuale Java non hanno ancora superato in efficienza i programmi compilati in modo nativo sia per un ritardo nello sviluppo delle tecnologie di compilazione adattive sia per una scelta troppo a basso livello del linguaggio della macchina virtuale (*bytecode*).

Scelte arbitrarie

Nessuna delle tre scelte arbitrarie descritte qui di seguito può essere colta come tale ragionando nella tradizione dei linguaggi imperativi.

- In generale è arbitrario che una funzione abbia un inizio ed una fine propri. La scelta è però ragionevole per l'architettura von Neumann. Quando una funzione inizia in genere deve fare delle operazioni per mappare i parametri formali su quelli attuali; in alternativa dovrebbe essere il chiamante a fare questo lavoro ma questo legherebbe la compilazione del chiamante e del chiamato. La fine di una funzione è in genere marcata da una apposita istruzione che trasferisce il controllo ad un indirizzo memorizzato nello stack; in alternativa il chiamante potrebbe memorizzare nello stack l'indirizzo corrispondente alla fine della sequenza che vuole chiamare ma questo complicherebbe in parte il compito del processore.

- Quando una funzione termina, il controllo viene trasferito all'istruzione successiva alla chiamata che ha provocato l'esecuzione della funzione stessa. L'esecuzione cioè continua in punti diversi a seconda del chiamante che pertanto deve fissare la continuazione. La scelta di fare di una chiamata a funzione sia un punto di partenza che di ritorno, è ragionevole per l'architettura von Neumann ma in generale è arbitraria.

- Infine, la scelta di far passare i parametri cioè di fare circolare i dati tra le funzioni è arbitraria in generale. Nel caso specifico della programmazione imperativa comunque non costituisce un problema perché in rappresentanza delle strutture dati circolano solo i puntatori.

Sovraspecificazioni obbligate

I linguaggi imperativi obbligano il programmatore a sovraspecificare i programmi nei modi che seguono. Per comprendere pienamente che si tratta di sovraspecificazioni è necessario porsi dal punto di vista di un'altra tradizione di programmazione che non richieda di specificare questi aspetti di un programma.

- È sempre necessario fissare la sequenza delle istruzioni anche quando non si è interessati all'ordine esatto di esecuzione.

- Anche se il nostro scopo, nella programmazione, è produrre dei valori, non possiamo limitarci a questi, dobbiamo anche occuparci delle celle di memoria dove risiedono (assegnare nomi, allocarle, …).

- È necessario definire la struttura dei dati che modellano un'entità complessa. Questa sovraspecificazione si manifesta principalmente nel dover definire la composizione degli aggregati.

- Quando in una funzione ne chiamo una seconda lo faccio solo perché sono interessato al risultato che calcola eppure sono tenuto a procurarmi (calcolare o farmi passare) tutti i parametri richiesti dalla funzione chiamata anche se non sono funzionali al mio algoritmo e non avrei bisogno di vincolarli.

Contributi all'Obiettivo di Adattabilità

Il paradigma imperativo affida la responsabilità della determinazione dei parametri attuali interamente al chiamante che pertanto li deve passare tutti contestualmente alla chiamata. Posto che non è supportata la ripartizione della responsabilità ci chiediamo in questo paragrafo in che misura sia consentita.

L'unico modo che ho per poter chiamare una funzione con un sottoinsieme dei parametri che richiede è definire una nuova funzione per ogni sottoinsieme di parametri. Per ogni funzione di *n* parametri devo quindi definire 2^n-1 sottofunzioni più *n* funzioni di ricerca dei parametri. L'alternativa di passare sistematicamente come parametro un array di coppie nome/valore è un modo implicito per riconoscere l'inadeguatezza del meccanismo di passaggio dei parametri.

Ogni sottofunzione deve chiamare la funzione completa usando al posto dei parametri attuali mancanti le corrispondenti funzioni di ricerca. Non è detto che il linguaggio mi consenta di definire due o più funzioni con lo stesso nome e comunque se ci sono dei parametri dello stesso tipo è necessario definire dei nuovi tipi specializzati o cambiare il nome della funzione. Inoltre anche le funzioni originali con tutti i parametri devono essere scritte come se ne mancassero per poterli aggiungere in seguito.

Una funzione di ricerca di un parametro ha a disposizione gli altri eventuali parametri attuali passati e deve provare a chiamare tutte le funzioni che possono produrre quel parametro a meno che non sia disponibile un valore passato separatamente dalla chiamata a funzione. Ci sono diverse complicazioni.

Ogni funzione di ricerca di un parametro ha bisogno di una lista di tutte le funzioni che possono produrre quel parametro con i relativi metodi per aggiungere/eliminare delle funzioni. Per ogni funzione è sufficiente inserire nella lista la sottofunzione che usa il maggior numero dei parametri attuali disponibili.

Ogni funzione di ricerca di un parametro ha bisogno di essere affiancata da una funzione che setti un valore per quel parametro e da una variabile per contenerlo. La funzione che setta il valore pone problemi di interferenza tra clienti che vogliono passare separatamente un parametro. Ma non è tutto; questo valore passato separatamente deve avere una validità limitata nel tempo altrimenti la funzione di ricerca non saprebbe decidere se fermarsi o proseguire. Per risolvere questo problema è necessario aggiungere una variabile booleana

che tenga traccia della disponibilità e una ulteriore funzione che comunichi la fine disponibilità del valore precedentemente passato. La soluzione semplificata di considerare monouso la disponibilità del valore ridurrebbe fortemente l'espressività del meccanismo. D'altra parte la funzione aggiunta complica il protocollo da usare per un corretto funzionamento e rende eccessivamente fragile il meccanismo.

La funzione di ricerca di un parametro così come l'ho descritta finora può dar luogo a ricorsione infinita. Per evitare questo problema è necessario aggiungere ad ogni funzione di ricerca un parametro che contenga l'insieme delle funzioni di ricerca già provate. Il risultato è una funzione di ricerca esaustiva con backtracking che termina con un fallimento se mancano dei parametri. Se si vuole supportare, ed è ragionevole, un passaggio separato dei parametri in modo concorrente, ammettendo quindi la possibilità che i parametri arrivino dopo la chiamata, è necessario sostituire il fallimento con una nuova ricerca.

La soluzione delineata è estremamente complessa: è improponibile come stile di programmazione ma è anche improponibile come implementazione nascosta dietro ad un precompilatore. È difficile da usare, è troppo fragile, è costosa in spazio e in tempo di esecuzione e non risolve neppure tutti i problemi che sono emersi come ad esempio le interferenze tra parametri separati.

La programmazione imperativa non supporta né consente una soluzione accettabile al problema posto.

Conclusioni

Tutti i punti deboli evidenziati derivano dalla scelta di legare strettamente i linguaggi imperativi all'architettura von Neumann. Questo legame ha d'altra parte rappresentato un vantaggio competitivo rispetto agli altri paradigmi analizzati in questa parte perché ha permesso di realizzare facilmente implementazioni efficienti.

L'esecuzione di un programma imperativo è dominata dal flusso del controllo. I dati hanno esclusivamente un ruolo passivo: non si spostano mai verso quello che possono contribuire a calcolare generando un flusso dei dati; stanno fermi in attesa che un'istruzione li venga a prendere o modificare. Una chiamata a funzione non origina mai una ricerca da parte della funzione chiamata dei modi per calcolare i parametri attuali, perché questi ultimi vengono sempre passati tutti contestualmente alla chiamata rendendo non necessario oltre che impossibile un flusso autonomo della domanda. Questo significa che il programmatore deve fare mentalmente una ricerca diretta da quello che vuole ottenere come risultato, scegliere tutto l'albero di funzioni da usare e se rimangono dei parametri scoperti deve calcolarli o farseli passare. Il programmatore deve cioè sovraspecificare una chiamata a funzione: non può limitarsi a dire *cosa* vuole ottenere, deve preoccuparsi anche di scrivere *come* farlo. Questo pregiudica la possibilità di aggiungere ad un programma esistente nuovi modi per ottenere un risultato.

I linguaggi imperativi non contribuiscono in nessun modo al raggiungimento dell'Obiettivo di Adattabilità.

Capitolo 16

PROGRAMMAZIONE FUNZIONALE

Questo capitolo esamina il paradigma di programmazione dei linguaggi che nascono dalla tradizione delle funzioni matematiche.

I linguaggi imperativi – descritti nel capitolo precedente – enfatizzano uno stile di programmazione in cui i programmi eseguono comandi sequenzialmente, usano variabili e le aggiornano con istruzioni di assegnamento. Al contrario, i linguaggi funzionali (puri) non hanno variabili, non hanno istruzioni di assegnamento e non hanno costrutti iterativi.

L'essenza della programmazione funzionale è la combinazione di funzioni al fine di produrre funzioni più potenti.

Funzioni matematiche

Una *funzione matematica* è una regola per trasformare o associare i membri di un insieme (dominio) in quelli di un altro (codominio). Una definizione di funzione specifica il dominio, il codominio e la regola di trasformazione per la funzione. Una volta che una funzione è stata definita, si può applicare ad un particolare elemento del dominio; questo elemento, detto argomento della funzione o parametro attuale, si sostituisce ad ogni occorrenza del parametro formale nella definizione e l'applicazione produce il corrispondente elemento del codominio. Un parametro formale è una variabile matematica che indica nella definizione della funzione un elemento generico dell'insieme dominio; nell'applicazione viene sostituito da un (solo) valore e in seguito non viene più modificato.

Trasparenza referenziale

Un sistema possiede *trasparenza referenziale* se il significato del tutto si può determinare esaminando solamente il significato delle parti immediatamente costituenti. Una espressione matematica gode di questa proprietà perché il suo valore dipende solo dai valori delle (eventuali) espressioni immediatamente costituenti. La sola cosa che importa di una espressione matematica è il valore che denota, e ogni sottoespressione può essere sostituita da ogni altra di ugual valore; inoltre il valore di una espressione è sempre lo stesso a parità di parametri. Per esempio, l'espressione matematica $f(x) + g(x)$ possiamo valutarla come $g(x) + f(x)$ oppure possiamo sostituire a f un'altra funzione f' se sappiamo che produce gli stessi valori di f; analogamente l'espressione matematica $f(x) + f(x)$ è uguale a $2f(x)$. In un linguaggio imperativo non possiamo fare queste sostituzioni senza analizzare il corpo delle funzioni f e g perché il significato dell'espressione potrebbe dipendere dalla storia della computazione delle sottoespressioni. Le istruzioni di assegnamento, i parametri passati per indirizzo e le variabili globali sono le ragioni principali per cui i linguaggi imperativi non godono di trasparenza referenziale. Nei linguaggi funzionali, un nome sta sempre per la stessa cosa pertanto godono della trasparenza referenziale.

Il modello di computazione funzionale

Descrivo il modello computazionale che più si avvicina al modello concettuale della programmazione funzionale [PeyLes92]. Un programma funzionale viene eseguito valutando una espressione rappresentata da un grafo. La valutazione consiste in una sequenza di riduzioni. Una *riduzione* sostituisce una espressione riducibile (*redex*) nel grafo con la sua forma ridotta. Una espressione è in *forma normale* quando non ci sono più redex. Ad ogni istante ci possono essere più redex nell'espressione da valutare e bisogna scegliere tenendo presente che indipendentemente dall'ordine di valutazione si arriva alla forma normale tranne che con alcune sequenze di riduzione che non terminano. D'altra parte se esiste una sequenza di redex che fa terminare la valutazione, allora terminerà anche con la politica di scegliere sempre il redex più esterno (outermost).

Strategie di valutazione

Una strategia di valutazione definisce se e in che ordine valutare i parametri attuali di una funzione. Tutte le strategie di valutazione si possono definire a partire da due che rappresentano i casi limite: valutazione stretta e valutazione su richiesta.

Nella *valutazione stretta* (*strict evaluation*) di una chiamata ad una funzione prima vengono valutati i parametri attuali poi vengono legati ai parametri formali dal primo all'ultimo e solo a questo punto viene valutato il corpo della funzione. Se un parametro attuale contiene una chiamata a funzione questa viene invocata quando viene valutato il parametro.

Nella *valutazione su richiesta* (*demand-driven* o *lazy evaluation* per usare la terminologia funzionale) di una chiamata ad una funzione viene subito iniziata la valutazione del corpo della funzione e ciascun parametro viene valutato solo se e quando strettamente necessario.

Anatomia dei linguaggi funzionali

Un linguaggio funzionale è costituito da quattro componenti: un insieme di funzioni primitive, un insieme di forme funzionali, l'operazione di applicazione e un insieme di strutture dati. Le funzioni primitive sono predefinite dal linguaggio e applicabili direttamente. Le forme funzionali sono un meccanismo che serve per combinare funzioni e crearne di nuove. L'operazione di applicazione è il meccanismo predefinito per applicare una funzione ai suoi argomenti e ottenere un risultato. Le strutture dati rappresentano tutti i possibili elementi degli insiemi dominio e codominio di ogni funzione. Per comodità è inoltre possibile assegnare un nome alle nuove funzioni definite.

Funzioni

Una funzione è una espressione con un inizio ed una fine propri; richiede in ingresso un elenco di parametri (formali) ed eventualmente produce in uscita un valore. L'applicazione di una funzione richiede il passaggio contestuale di tutti i

parametri (attuali) e si conclude con la sostituzione dell'applicazione con il valore risultato. Le funzioni sono spesso definite distinguendo diversi casi, ognuno viene definito separatamente applicando altre funzioni eventualmente in modo ricorsivo.

Funzioni di ordine superiore

Si dice *forma funzionale* o *funzione di ordine superiore* una funzione che richieda almeno una funzione per argomento o che produca una funzione come risultato. Gli strumenti che consentono di costruire nuove funzioni a partire da altre più semplici sono funzioni di ordine superiore. Ad esempio la composizione di funzioni è una funzione di ordine superiore che prende due funzioni come parametri e produce la funzione equivalente all'applicazione di una al risultato dell'altra.

Una funzione passata come argomento ad un'altra funzione può servire per scrivere funzioni flessibili e funzioni di mappatura. Una *funzione flessibile* implementa l'algoritmo generico e si fa passare le funzioni specifiche per adattarlo a diverse situazioni. Una *funzione di mappatura* definisce uno schema per rendere una funzione definita per uno o due argomenti applicabile ad una intera struttura dati. Ad esempio una funzione di ordinamento può avere come parametro la funzione di confronto. Entrambi gli usi si possono ottenere in un linguaggio orientato agli oggetti facendosi passare un oggetto che implementi un'interfaccia con le funzioni richieste.

Produrre una funzione come risultato può servire per fare composizione di funzioni, chiusure e liste infinite. Il linguaggio Haskell [PeyHug99] ha una funzionalità che si chiama *currying* che facilita l'uso di funzioni di ordine superiore. Una funzione di n argomenti può essere chiamata con solo m argomenti, $m<n$; il risultato è una nuova funzione di $n-m$ argomenti con i primi m parametri legati.

Assenza di variabili e assegnamento

Una variabile è un identificatore legato ad un valore che può essere cambiato durante l'esecuzione del programma. Non ci sono variabili in un linguaggio funzionale, però ci sono degli identificatori legati a dei valori. Una volta assegnato un valore ad un identificatore questo non viene più modificato. Gli identificatori in genere vengono legati ad un valore durante il passaggio dei parametri. Le variabili non sono necessarie in questo paradigma proprio perché il risultato di una funzione viene immediatamente passato come parametro ad un'altra funzione.

Un assegnamento distruttivo determina una dipendenza dal tempo e dalla sequenza nel significato di un nome. Proibire l'assegnamento distruttivo implica che un nome una volta legato ad un valore deve rimanervi legato fintanto che il nome è visibile. Pertanto il significato di un nome è costante e non dipende dall'ordine di esecuzione.

Anche le strutture dati coerentemente con questa scelta possono essere modificate solo in modo non distruttivo cioè producendo una nuova occorrenza che differisce dalla precedente per il solo cambiamento fatto.

Assenza di iterazioni

La ripetizione di sezioni di codice è fondamentale nella programmazione. L'assenza di iterazioni nella programmazione funzionale è una conseguenza dell'assenza di assegnamento. Un ciclo esplicito richiede almeno una variabile o una espressione che muta valore. I cicli espliciti sono sostituiti dalla ricorsione di coda e da costrutti di iterazione implicita.

La ricorsione di coda è una chiamata ricorsiva ad una funzione fatta subito prima della fine – non necessariamente lessicale – della funzione. Da sola la ricorsione di coda produce lo stesso effetto di un ciclo infinito; combinata assieme ad una condizione può simulare il comportamento di un ciclo *while*, *repeat* o *for*. La variabile di ciclo della versione iterativa viene sostituita da un

parametro nella versione ricorsiva. Ad ogni chiamata ricorsiva, il parametro attuale viene assegnato con la stessa espressione di incremento del ciclo.

I meccanismi iterativo e ricorsivo per realizzare le ripetizioni, pur esibendo lo stesso comportamento, sono molto differenti a livello concettuale (del modello di esecuzione). Nell'esecuzione iterativa di un ciclo, ad ogni ripetizione, viene cambiato solo il valore della cella di memoria che contiene la variabile di ciclo e quando l'espressione di fine ciclo viene soddisfatta l'esecuzione passa immediatamente all'istruzione successiva. Nell'esecuzione ricorsiva di un ciclo, ad ogni ripetizione, viene sostituita l'applicazione ricorsiva con il corpo della funzione cioè si espande l'espressione che contiene il ciclo; quando la condizione di fine ciclo viene soddisfatta vengono eseguite tutte le riduzioni e il valore di ritorno viene sostituito al corpo della funzione cominciando da quella più annidata.

Concettualmente la ricorsione esegue i cicli usando più spazio e più tempo. A livello implementativo il compilatore riconosce la ricorsione di coda e la compila esattamente come si fa con i cicli iterativi.

I costrutti di iterazione implicita applicano iterativamente una funzione a tutti i componenti di un oggetto composto coerente (array, lista). Per essere utile un costrutto iterativo in un linguaggio funzionale deve ritornare un valore perché è proibito l'assegnamento distruttivo per modificare l'oggetto e non vi è altro modo per comunicare il risultato dell'iterazione.

Critica del modello funzionale

L'assenza di effetti collaterali tipica della programmazione funzionale rende la semantica dei programmi (che terminano) indipendente dalle strategie di valutazione. L'ordine di valutazione delle espressioni e delle sotto espressioni è irrilevante. Quindi l'esecutore di un programma funzionale può scegliere liberamente la strategia e può anche cambiarla durante l'esecuzione. Anche i linguaggi imperativi possono adottare una valutazione su richiesta ma devono

specificarlo nel contratto con il programmatore e non è conveniente come scelta generale. Per esempio Java, C e C++ usano una strategia di valutazione su richiesta per gli operatori booleani nelle espressioni condizionali (se il primo termine di un operatore *and* è falso il secondo non viene valutato; analogamente se il primo termine di un operatore *or* è vero il secondo non viene valutato).

La valutazione su richiesta è interessante anche perché la valutazione stretta può calcolare più di quello che serve e metterci quindi più tempo. Poiché la determinazione automatica della strategia di valutazione preferibile è un problema aperto e difficile generalmente si lascia al programmatore la scelta della strategia da usare nella valutazione delle varie parti di un programma.

I linguaggi funzionali non obbligano il programmatore a fissare la sequenza delle operazioni da eseguire. L'ordine di valutazione è vincolato dalla dipendenza dei dati che il programmatore definisce implicitamente nelle espressioni e più esplicitamente nel passaggio dei parametri. Mentre in un linguaggio imperativo i *vincoli sequenziali* legano una istruzione da eseguire alla successiva nella direzione che va dall'inizio della funzione alla sua fine; in un linguaggio funzionale i vincoli sequenziali espressi dalla dipendenza dei dati legano una funzione nella direzione che va dall'ultima operazione che porta al risultato indietro fino alle funzioni passate come argomenti. Dirò pertanto che in un programma imperativo i vincoli sequenziali esprimono una *sequenza diretta* mentre in un programma funzionale una *sequenza inversa*.

Nella programmazione funzionale non esistono variabili e non si pone il problema della gestione delle celle di memoria; il programmatore può concentrarsi sulla produzione di valori. D'altra parte, l'immutabilità dei dati pone un nuovo problema di progettazione che distoglie nuovamente il programmatore dai propri obiettivi: il problema dell'*identità multipla.* Ogni cambiamento ad una struttura dati crea una nuova copia modificata; se esiste almeno un'altra parte del programma che continua ad usare la struttura dati (non modificata) allora quest'ultima si ritrova ad avere due identità. Quando una

struttura dati viene distribuita a più funzioni che pertanto possono modificarla separatamente mi devo preoccupare che la sua identità venga preservata nel senso richiesto dall'algoritmo.

Il problema dell'identità multipla è il simmetrico del problema dell'assegnamento distruttivo. Nella programmazione imperativa si assume che tutte le funzioni vogliano collaborare per mantenere un'unica identità di una struttura dati ricevuta come parametro e pertanto siano interessate di regola solo alla sua copia più aggiornata; di conseguenza, le modifiche possono essere eseguite direttamente. Se una funzione vuole legarsi ad una certa identità della struttura dati e/o svilupparla autonomamente, deve provvedere esplicitamente a farsi una copia (della profondità che vuole riservarsi). Se il linguaggio è concorrente si pone però il problema dell'atomicità delle modifiche pena interferenze non desiderate. Viceversa nella programmazione funzionale si assume che tutte le funzioni vogliano mantenere/sviluppare autonomamente l'identità di una struttura dati e pertanto siano interessate di regola solo all'identità che hanno ricevuto come parametro; di conseguenza, le modifiche vengono implicitamente precedute da una copia (profonda). Una funzione non può imporre una nuova identità per una struttura dati a tutte le altre funzioni; può proporre in uscita una nuova identità e può fare in modo che le funzioni da lei chiamate collaborino a definire una unica nuova identità fissando opportune dipendenze sui dati. L'idea base è che se due funzioni modificano una struttura dati e restituiscono la nuova, componendo le due funzioni ottengo una struttura dati con entrambe le modifiche. Una strategia di esecuzione concorrente di un programma funzionale non ha modo di produrre interferenze perché, come detto, la collaborazione per sviluppare una unica nuova identità di una struttura dati viene ottenuta introducendo dipendenze sui dati cioè vincoli sequenziali.

Nei programmi attuali l'obiettivo prevalente è (ancora) sviluppare un unico stato corrente piuttosto che mantenere una sorta di memoria storica degli stati attraversati dal sistema. Pertanto è comprensibile sia la scelta di alcuni linguaggi

funzionali di supportare anche entità mutabili sia la maggior popolarità dei linguaggi imperativi.

Scelte arbitrarie

Nessuna delle tre scelte arbitrarie descritte qui di seguito può essere colta come tale ragionando nella tradizione dei linguaggi funzionali.

- In generale è arbitrario che una funzione abbia un inizio ed una fine propri. La scelta è però ragionevole nella logica dalle funzioni matematiche. Per poter valutare *sqrt(abs(-4))* devo conoscere l'inizio e la fine delle due funzioni che vengono usate. Rispetto alle funzioni imperative che contengono una sequenza di istruzioni, le funzioni dei linguaggi funzionali possono contenere solo una espressione pertanto hanno di solito una granularità più fine che riduce gli effetti di questa scelta arbitraria.

- Quando una funzione termina, il valore calcolato viene sostituito alla chiamata che aveva iniziato il calcolo. L'esecuzione cioè continua in punti diversi a seconda del chiamante che pertanto implicitamente fissa la continuazione. La scelta di fare di una chiamata a funzione sia un punto di partenza che di ritorno, è comprensibile per analogia con le funzioni matematiche ma in generale è arbitraria.

- Infine, la scelta di far passare i parametri cioè di fare circolare i dati tra le funzioni è arbitraria in generale. Nella programmazione funzionale, questa scelta comporta concettualmente delle copie profonde delle strutture dati.

Sovraspecificazioni obbligate

I linguaggi funzionali obbligano il programmatore a sovraspecificare i programmi nei modi che seguono. Per comprendere pienamente che si tratta di sovraspecificazioni è necessario porsi dal punto di vista di un'altra tradizione di programmazione che non richieda di specificare questi aspetti di un programma.

- È necessario definire la struttura dei dati che modellano un'entità complessa. Questa sovraspecificazione si manifesta in particolare nel dover definire la composizione degli aggregati.

- Quando in una funzione ne chiamo una seconda lo faccio solo perché sono interessato al risultato che calcola eppure sono tenuto a procurarmi (calcolare o farmi passare) tutti i parametri richiesti dalla funzione chiamata anche se non sono funzionali al mio algoritmo e non avrei bisogno di vincolarli.

Contributi all'Obiettivo di Adattabilità

Il paradigma funzionale affida la responsabilità della determinazione dei parametri attuali interamente al chiamante che pertanto li deve passare tutti contestualmente alla chiamata. Posto che non è supportata la ripartizione della responsabilità tra chiamante e funzione chiamata ci chiediamo in questo paragrafo in che misura sia consentita.

Le funzioni di ordine superiore permettono di estendere esplicitamente una funzione per composizione o implicitamente per currying. Si tratta di una funzionalità interessante perché opera nella direzione della sequenza inversa cioè permette di costruire funzioni sempre più potenti per produrre uno stesso risultato; ma è diversa da quella richiesta per tre motivi interdipendenti: l'estensione è unica, estende funzioni non applicazioni di funzioni, la chiamata si lega ad una particolare funzione composta.

Per composizione o currying posso legare un argomento di una funzione ad una ed una sola altra funzione che può calcolarlo. Non posso legare un argomento a due o più funzioni che lo calcolano ed aspettarmi che quando viene richiesto il parametro attuale venga iniziata la valutazione (in modo concorrente o in un qualche ordine) di tutte le funzioni che possono calcolarlo e che questa valutazione proceda fintantoché non viene determinato in un qualche modo il parametro attuale.

Le funzioni di ordine superiore possono avere argomenti di tipo funzione e restituire una nuova funzione ma non possono prendere l'applicazione di una funzione ed estenderla legando un argomento ad un'altra funzione. Non è possibile cioè estendere una funzione dopo che la si è applicata. Ovvero dall'altro punto di vista, il chiamante ha la responsabilità di comporre la funzione che vuole chiamare. Le funzioni di ordine superiore sono un meccanismo in mano al chiamante; la funzione chiamata è sempre completamente determinata nel punto dell'invocazione.

Si può definire una nuova funzione che sia la composizione di due funzioni esistenti. In generale si può comporre un albero con tutte le funzioni che più o meno indirettamente possono contribuire a produrre i parametri attuali di una funzione data e si può assegnare un nome ad ogni suo sotto albero. In questo modo si definisce una base di conoscenze con tutti i modi per produrre un certo risultato. Un chiamante, in base a quello che vuole ottenere e ai dati di cui dispone, può scegliere la funzione più adatta; se quest'ultima gli richiede comunque di passare come parametro attuale un'espressione allora può definire per composizione nuove funzioni in modo da ridursi sempre al caso di passaggio di dati disponibili o di dati che a sua volta si fa passare. Questa strategia garantisce la massima crescita della base di conoscenze. Una chiamata a funzione però resta sempre una chiamata ad una particolare funzione e fissa tutti i parametri attuali compresi quelli che non dipendono dall'algoritmo della funzione chiamante.

La programmazione funzionale non supporta né consente una soluzione accettabile al problema posto; però fornisce soluzioni parziali a due sotto problemi. Le funzioni di ordine superiore permettono di estendere esplicitamente l'algoritmo di ricerca/produzione dei parametri attuali e, restituendo una nuova funzione composta, è come se definissero un contesto per i parametri della nuova funzione (che provengono da diverse funzioni).

L'assenza di puntatori sposta solo il problema. Il legame tra parametro formale e parametro attuale è soggetto agli stessi vincoli: percorso di ricerca singolo e con un inizio determinato. Una chiamata è obbligata a fornire uno ed un solo parametro attuale per ogni parametro formale della funzione chiamata.

Conclusioni

Anche l'esecuzione di un programma funzionale è dominata dal flusso del controllo. Però il motore inferenziale, grazie all'assenza di effetti collaterali, può scegliere la strategia di valutazione e privilegiare ad esempio il flusso della domanda (valutazione lazy). I dati possono avere un ruolo passivo e stare fermi in attesa che un'espressione li richieda oppure possono avere un ruolo attivo e propagarsi verso il risultato. In quest'ultimo caso si ha un flusso dei dati limitato al percorso scelto dal programmatore. Analogamente, una chiamata a funzione intesa come punto di ritorno può avere un ruolo passivo ed attendere che il flusso dei dati arrivi a produrre il risultato oppure può avere un ruolo attivo e propagare la richiesta verso i dati disponibili. In quest'ultimo caso si ha un flusso della domanda limitato al percorso scelto dal programmatore. In ogni caso il programmatore deve fare mentalmente una ricerca diretta dal risultato che vuole ottenere, scegliere tutto l'albero di funzioni da usare e se rimangono dei parametri scoperti deve calcolarli o farseli passare. Il programmatore deve cioè sovraspecificare una chiamata a funzione: non può limitarsi a dire *cosa* vuole ottenere, deve preoccuparsi anche di scrivere *come* farlo. Questo pregiudica la possibilità di aggiungere ad un programma esistente nuovi modi per ottenere un risultato.

I linguaggi funzionali contribuiscono a risolvere alcuni sotto problemi dell'Obiettivo di Adattabilità. In particolare le funzioni di ordine superiore sono un meccanismo per creare un contesto trasversale tra più funzioni che devono collaborare alla produzione di un risultato.

Capitolo 17

PROGRAMMAZIONE DATAFLOW

Il modello dataflow nasce da due tradizioni: macchine parallele e programmazione visuale. Il fine della prima è realizzare architetture parallele non von Neumann e linguaggi di programmazione che le sfruttino esplicitando il parallelismo a livello fine. Il fine della seconda tradizione è realizzare linguaggi di programmazione più semplici da programmare usabili anche da utenti non programmatori.

I fogli di calcolo incorporano una forma di computazione diretta dai dati per aggiornare i valori delle celle che dipendono da altri valori. La rappresentazione interna di un foglio di calcolo è molto simile al grafo di un programma dataflow. Ogni cella del foglio di calcolo è programmata separatamente. Per una cella che fa da foglia del grafo dataflow, il programma indica il valore di questa cella. Per una cella nodo di computazione, il programma indica come ricalcolare il valore di questa cella in base al valore di altre celle. Quando viene cambiato il valore di una cella il nuovo valore viene propagato a tutte le celle dipendenti che necessitano di essere ricalcolate.

In un linguaggio dataflow un programma non è rappresentato da un sequenza lineare di istruzioni, ma da un grafo. Il grafo rappresenta l'ordine parziale della sequenza da valutare; in contrasto, un tipico linguaggio von Neumann crea una sequenza di istruzioni totalmente ordinata.

In un programma dataflow non esiste un singolo thread di esecuzione che procede da una istruzione alla successiva domandando dati, eseguendo un'operazione e restituendo nuovi dati (control-driven). Al contrario i dati

fluiscono alle istruzioni causando la valutazione non appena tutti gli operandi sono disponibili (data-driven).

Il grafo dataflow non include la nozione di variabili. I valori scorrono sugli archi del grafo: per i tipi semplici non ci sono problemi; i tipi strutturati, il tipo puntatore ed il tipo funzione si adattano meno bene al formalismo dataflow. Non esiste neppure un semplice concetto di funzione che ritorni un valore. Si può immaginare di prendere parte del grafo, chiamare gli archi entranti parametri e quelli uscenti risultati; ma questa organizzazione porta a definizioni ricorsive come vedremo.

Il modello di computazione dataflow

Il modello di computazione dataflow non è basato sul flusso del controllo ma sul flusso dei dati. In contrasto con il modello di computazione von Neumann, l'esecuzione di una operazione (istruzione) si basa sulla disponibilità degli operandi piuttosto che su una sequenza predefinita.

In un sistema diretto dal flusso dei dati, il compilatore genera un grafo di dipendenze che descrive come un'istruzione dipenda dal risultato di altre istruzioni. Concettualmente un programma dataflow è un grafo orientato che può contenere cicli. I nodi del grafo denotano operazioni, gli archi denotano dipendenze tra operazioni. Un nodo viene eseguito quando tutti i suoi ingressi sono disponibili. L'azione di un nodo dipende solo dai valori degli ingressi e non ha effetti collaterali (side effects).

Anatomia delle architetture dataflow

I sistemi dataflow differiscono tra loro principalmente per il diverso supporto al codice rientrante (static vs dynamic), alle funzioni (data-driven vs demand-driven), e alle strutture dati (centralized vs distributed). In comune hanno una rappresentazione a grafo dei programmi. Per un'ampia rassegna e confronto delle architetture dataflow si veda [Jag95, SneEga94].

Grafo dataflow

Un grafo dataflow è un grafo orientato in cui i nodi denotano operazioni e gli archi denotano dipendenze tra le operazioni denotate dai nodi. I valori prodotti e consumati dai nodi vengono conservati in token che scorrono lungo gli archi. In aggiunta ai nodi per eseguire operazioni aritmetiche, relazionali e logiche sono necessari dei nodi di controllo per esprimere oltre alle espressioni anche cicli e condizioni.

Supporto al codice rientrante

Il codice rientrante: invocazione multipla, ricorsione e cicli, nasconde del parallelismo. I modelli di esecuzione statico e dinamico si distinguono per la scelta da parte di quest'ultimo di esplicitare il parallelismo dinamico.

Nel modello statico un operatore denotato da un nodo è eseguibile quando in ognuno degli archi entranti è presente un token (valore). L'esecuzione di un nodo causa la consumazione dei token in ingresso e la produzione di un token nell'arco uscente. Ogni arco può contenere solo un token per volta. Quindi un operatore eseguibile può essere effettivamente eseguito solo quando il suo arco uscente è vuoto.

Il modello di computazione statico può esplicitare solo il *parallelismo strutturale* che consiste nella esecuzione simultanea di diverse operazioni non correlate. Poiché ogni arco può contenere solo un token per volta solo una iterazione o una invocazione di funzione può essere attiva. Pertanto non può esplicitare le forme di *parallelismo dinamico* come il parallelismo nei cicli (esecuzione simultanea di diverse iterazioni non correlate del corpo di un ciclo) o il parallelismo nella ricorsione (esecuzione simultanea di più chiamate ricorsive ad una funzione).

Nel modello dinamico ogni token ha associato un tag che contiene informazioni aggiuntive (identificatore di invocazione, di iterazione, …) che lo lega ad un contesto; un operatore denotato da un nodo è eseguibile quando in ognuno degli archi entranti è presente un token con i tag identici. L'esecuzione di un nodo causa la consumazione dei token in ingresso usati e la produzione di un

token nell'arco uscente con tag appropriato. Il confronto dei tag è costoso e richiede memoria associativa; inoltre il modello esplicita eccessivamente il parallelismo e necessita di meccanismi per limitarlo superiormente (loop bounding).

Supporto alle strutture dati

Il modello di computazione dataflow, come detto, è basato sul flusso dei dati. Mentre nel modello object-oriented, gli oggetti sono stazionari e si scambiano messaggi; al contrario, nel modello dataflow, le operazioni sono stazionarie e i dati scorrono sugli archi del grafo dataflow da una operazione ad un'altra. I tipi primitivi (interi, booleani, caratteri, …) si prestano naturalmente a questo modello mentre i tipi strutturati, il tipo puntatore ed il tipo funzione pongono dei problemi. I primi linguaggi dataflow e anche alcuni dei più recenti come Java Studio evitano il problema non supportando le strutture dati. Gli altri usano due approcci: stream o stato.

I primi applicano uniformemente il modello di computazione dataflow a tutti i dati compresi quelli strutturati. Questi ultimi scorrono nel grafo dataflow sotto forma di stream di dati omogenei nel caso di array e in genere non omogenei per i record. Sperimentalmente è stato osservato [Sne93] che, se eseguito su architetture parallele, il peso di dover portare in giro intere strutture dati viene compensato dal miglior sfruttamento del parallelismo dovuto all'assenza di effetti collaterali legati allo stato.

I secondi reintroducono una nozione esplicita di stato. Gli array, le strutture dati e le funzioni risiedono ad un particolare indirizzo e vengono rappresentati nel grafo dataflow da puntatori che scorrono senza problemi come gli altri tipi primitivi. In questi sistemi basati su stato le strutture dati sono mantenute come oggetti globali pertanto tutti gli accessi appaiono come se fossero fatti ad una memoria condivisa globalmente. Questa soluzione è adeguata per l'architettura von Neumann ma scala con gli stessi problemi (colli di bottiglia) quando si cerca di adattarla ad architetture parallele.

Supporto alle funzioni

Per definire una funzione si può immaginare di prendere una parte del grafo dataflow e chiamare gli archi entranti parametri e quelli uscenti risultati; rispetto ad una definizione classica di funzione vi sono due differenze rilevanti: l'inizio esecuzione e il chiamante.

Una funzione in un linguaggio imperativo è una sequenza di istruzioni con un (solo) inizio ed una fine; i parametri sono tutti disponibili quando inizia l'esecuzione e vengono passati dal chiamante. Una funzione dataflow è un albero (in generale un grafo) con tanti inizi quanti sono i parametri; ognuno entra ad un certo punto della funzione e da lì in poi partecipa alla determinazione del risultato. Ogni parametro attuale di una funzione può iniziarne l'esecuzione, le uniche restrizioni sono date dalle dipendenze dagli altri parametri. Alcuni sistemi dataflow consentono di sincronizzare i parametri in ingresso (eventualmente solo alcuni) in modo da ricondursi alla semantica di una funzione classica.

Ha senso definire una funzione se poi posso usarla in più parti del programma. Le diverse chiamate ad una stessa funzione devono essere indipendenti durante l'esecuzione. In particolare le chiamate concorrenti non devono interferire nel passaggio dei parametri e la terminazione della funzione chiamata deve far riprendere l'esecuzione solo del proprio chiamante. La semplice definizione di funzione dataflow riportata sopra non soddisfa nessuno dei due requisiti. Non viene data nessuna garanzia sulla provenienza dei singoli parametri e in più il risultato attiva le continuazioni di tutti i chiamanti.

Il fatto che il risultato di una operazione denotata da un nodo attivi tutti gli archi uscenti cioè tutto ciò che si può calcolare a partire da quel risultato senza tenere conto di ciò che si vuole calcolare porta nella migliore delle ipotesi a produrre token superflui. Nel modello di *esecuzione diretta dalla domanda* una operazione denotata da un nodo viene eseguita solo quando viene richiesto un particolare token sull'arco uscente e i necessari token sono disponibili sugli archi entranti.

Se questi ultimi non sono disponibili, la domanda si propaga all'indietro lungo gli archi entranti. I token vengono prodotti su un arco solo quando vengono esplicitamente richiesti; in questo modo nessun token superfluo viene prodotto. La domanda si propaga all'indietro lungo gli archi del grafo dataflow: non vengono definiti nuovi archi.

Il modello di esecuzione dataflow diretto dai dati può esser visto come un caso particolare di quello diretto dalla domanda con domande implicite sempre presenti. D'altra parte il modello di esecuzione dataflow diretto dalla domanda può essere simulato da quello diretto dai dati aumentando il grafo dataflow originale con il grafo corrispondente alla propagazione della domanda. In ogni caso, tener conto della domanda risolve solo metà del problema della definizione di funzioni mentre procedendo per un'altra strada si può risolvere tutto il problema.

L'idea base usata per risolvere interamente il problema è replicare il corpo della funzione per ogni chiamante mantenendo l'unità della funzione solo a livello di ambiente di programmazione. Questa soluzione viene generalmente presentata in forma gerarchica. Per ogni funzione viene definito un nuovo tipo di nodo da usare quando si vuole applicare la funzione. Questo porta ad avere un grafo dataflow gerarchico ovvero una foresta di grafi dataflow. Inoltre come ulteriore ottimizzazione si può replicare solo lo stato di esecuzione (runtime) anziché tutto il corpo di una funzione.

In entrambi i casi la soluzione funziona ma è concettualmente incoerente e incompleta. Se replico devo spiegare come mantenere unitarie le copie; se uso nodi gerarchici ne devo spiegare il funzionamento in termini dataflow. L'aggiunta della domanda avrebbe consentito una soluzione corretta seppur molto conservativa.

Sintassi dei linguaggi dataflow

Un sorgente testuale è inadeguato per rappresentare il grafo di un programma dataflow. Ciò nonostante la quasi totalità dei linguaggi dataflow hanno una

sintassi testuale che per di più si ispira a quella dei linguaggi funzionali. Talvolta addirittura, il modello dataflow è usato solo come modello di esecuzione per i linguaggi funzionali. Di conseguenza, la programmazione in questi linguaggi non permette di cogliere nessuna delle peculiarità del modello dataflow.

Per rappresentare un grafo dataflow è necessaria una interfaccia di programmazione visuale. Linguaggi come Java Studio e Prograph permettono di apprezzare i vantaggi del modello dataflow.

Java Studio

Java Studio è un linguaggio di programmazione dataflow visuale basato su componenti Java (JavaBeans). Permette di creare Java applets, applicazioni e componenti JavaBeans. E' particolarmente indicato per realizzare velocemente contenuto Java da inserire in pagine web. Consente di definire delle funzioni nella forma di nuovi componenti riusabili. Non supporta le strutture dati né l'allocazione dinamica.

Prograph

Prograph è un linguaggio di programmazione dataflow orientato agli oggetti completamente visuale. Usa connessioni per esprimere il flusso del controllo (synchro). Nella programmazione object-oriented gli oggetti sono stazionari e si scambiano messaggi; in Prograph gli oggetti fluiscono nei nodi che denotano metodi.

Critica del modello dataflow

La maggior parte dei linguaggi dataflow sono nati nell'ambito della ricerca sulle macchine parallele con una conseguente eccessiva enfasi sul parallelismo e una scelta di fatto quando non esplicita di non porsi in concorrenza con gli altri linguaggi sull'architettura von Neumann. Per lo stesso motivo, il modello dataflow è stato sviluppato prevalentemente come architettura; anche nelle implementazioni per i computer tradizionali si è preferito realizzare delle

macchine virtuali dataflow sulle quali far girare linguaggi funzionali poco o per nulla adattati al modello dataflow.

Java Studio e Prograph nascono dalla tradizione dei linguaggi visuali con l'intento di semplificare la programmazione. L'enfasi sulla facilità ha portato a realizzare linguaggi del tutto simili a quelli imperativi orientati agli oggetti con la sola eccezione che il corpo delle funzioni è dataflow e programmabile in modo visuale. Perlomeno la rappresentazione visuale permette di cogliere, seppur nella brevità del corpo di una funzione, il flusso diretto dai dati.

Nessuna delle due tradizioni in cui sono nati i linguaggi dataflow ha pertanto esplorato realmente le potenzialità del modello. Molte scelte arbitrarie e sovraspecificazioni obbligate rilevate anche negli altri paradigmi qui appaiono in tutta la loro evidenza.

Il modello dataflow non obbliga il programmatore a specificare la sequenza esatta delle istruzioni quando l'algoritmo non lo richieda. Il grafo dataflow infatti rappresenta anche l'ordine di valutazione imposto dalla dipendenza dei dati (data-dependency graph). Ed è sottoposto al vincolo che un'espressione non può essere valutata prima della valutazione dei propri operandi. Un'analisi di tipo dataflow sulla sequenza delle istruzioni permette di ricostruire in parte il grafo di dipendenza dei dati. I compilatori dei linguaggi imperativi eseguono questo tipo di analisi per trovare un ordine di esecuzione più efficiente delle istruzioni e per esplicitare il parallelismo. Si può anzi dire che l'analisi dataflow sia il maggior contributo dei linguaggi dataflow.

Nella programmazione dataflow non esistono variabili e non si pone il problema della gestione delle celle di memoria; il programmatore può concentrarsi sul flusso dei dati. D'altra parte, a distogliere nuovamente il programmatore dai propri obiettivi c'è il problema dell'identità multipla già descritto per i linguaggi funzionali e che qui è ancora più evidente. Le ramificazioni del flusso dei dati comportano la creazione di copie che possono essere modificate separatamente

dando luogo ad altrettante identità; il programmatore ha il compito di preservare l'identità dei dati nel senso richiesto dagli algoritmi del programma.

Scelte arbitrarie

Le prime tre scelte descritte qui di seguito possono essere riconosciute come arbitrarie ragionando nel modello dataflow; per trovare alternative ragionevoli è però necessario non farsi influenzare dalla tradizione delle funzioni matematiche. La quarta scelta arbitraria non può essere colta come tale ragionando nella tradizione dei linguaggi dataflow.

- È arbitrario fissare dei nodi di un grafo dataflow come inizio e fine propri di una funzione. È il chiamante che sa cosa vuole ottenere e di quali dati dispone; pertanto è sua responsabilità delimitare con degli archi la parte di grafo che calcola la funzione a cui è interessato. Chiamanti diversi inoltre possono avere l'esigenza di definire funzioni parzialmente sovrapposte sul grafo dataflow. I problemi di interferenza tra le chiamate e tra le continuazioni, già ampiamente discussi, hanno contribuito in modo determinante a ricondurre forzatamente le soluzioni alla più famigliare definizione di funzione.

- È arbitrario stabilire che quando una funzione termina, l'esecuzione proceda con la continuazione del chiamante. La scelta di fare di una chiamata a funzione sia un punto di partenza che di ritorno, è comprensibile per analogia con le funzioni matematiche ma è una forzatura per il modello dataflow. Infatti il chiamante porta avanti i dati per iniziare la funzione mentre la continuazione a cui viene arbitrariamente legato porta avanti il risultato della funzione. Inoltre il risultato può utilmente essere portato avanti anche da altre continuazioni.

- I parametri di una funzione provengono da altrettanti flussi indipendenti che li calcolano; se già appariva una forzatura riunire nella chiamata la partenza e

il ritorno a maggior ragione si può cogliere l'arbitrarietà di obbligare il chiamante a passare contestualmente alla chiamata anche tutti i parametri.

- La scelta di far circolare i dati sugli archi del grafo è arbitraria in generale. Nella programmazione dataflow, questa scelta appare inevitabile e comporta concettualmente delle copie profonde delle strutture dati ad ogni diramazione.

Sovraspecificazioni obbligate

I linguaggi dataflow obbligano il programmatore a sovraspecificare i programmi nel modo che segue. Per comprendere pienamente che si tratta di una sovraspecificazione è necessario porsi dal punto di vista di un'altra tradizione di programmazione che non richieda di specificare questo aspetto di un programma.

- È necessario definire la struttura dei dati che modellano un'entità complessa. Questa sovraspecificazione si manifesta in particolare nel dover definire la composizione degli aggregati.

Contributi all'Obiettivo di Adattabilità

Il paradigma dataflow affida la responsabilità della determinazione dei parametri attuali interamente al chiamante che pertanto li deve passare tutti contestualmente alla chiamata. Posto che non è supportata la ripartizione della responsabilità tra chiamante e funzione chiamata ci chiediamo in questo paragrafo in che misura sia consentita.

Nel paradigma dataflow non è supportata la possibilità di definire un flusso della domanda di conseguenza la funzione chiamata non può far altro che attendere passivamente l'arrivo di tutti i dati. D'altra parte, una cosa è attribuire la responsabilità al chiamante un'altra è localizzare questa responsabilità nella chiamata. I parametri di una funzione provengono da altrettanti flussi indipendenti che li calcolano; solo con una forzatura al modello si è fatta la scelta sintattica di far passare i flussi dei parametri e il flusso di ritorno per una

strozzatura come la chiamata a funzione. Semanticamente una chiamata a funzione serve per creare un contesto comune ai parametri e al risultato; cioè serve ad impedire che si verifichino delle interferenze tra chiamate diverse nel passaggio dei parametri e a limitare la prosecuzione alla continuazione del chiamante.

Fornire i parametri è quindi una responsabilità distribuita fra tutte le funzioni che possono produrli. Il collegare il risultato di una nuova funzione ad un parametro di una funzione esistente equivale ad estenderla; è possibile collegare diverse funzioni ad uno stesso parametro attuale. Per fare questa operazione anche dinamicamente è necessario che il linguaggio sia riflessivo. L'introduzione di una chiamata a funzione deve essere accompagnata da una operazione di copia della parte del grafo dataflow che va dai parametri formali della nuova funzione fino al suo risultato. Questa operazione è analoga all'applicazione di una funzione di ordine superiore ma richiede meno informazioni. Infatti è sufficiente precisare gli ingressi e l'uscita mentre la composizione di funzioni richiede di specificare tutte le funzioni coinvolte indicando anche l'ordine di composizione.

Rispetto al paradigma funzionale nel modello dataflow posso definire con una modalità più incrementale un grafo di funzioni composte usabile come un database di conoscenze. Una chiamata a funzione però resta sempre una chiamata ad una particolare funzione e fissa tutti i parametri attuali compresi quelli che non dipendono dall'algoritmo della funzione chiamante. Questo problema può essere risolto solo rinunciando a definire in modo unitario la chiamata a funzione.

L'assenza di un flusso diretto dalla domanda che possa assumersi la responsabilità di reperire i dati obbliga il modello di esecuzione a farli scorrere sul grafo.

Un modello di esecuzione basato unicamente sul flusso dei dati porta a calcolare tutto ciò che è derivabile; il calcolo in generale è molto di più di quello che viene

richiesto dal programma. Pertanto è opportuno limitare l'avanzamento dei dati all'effettiva richiesta introducendo sullo stesso grafo un flusso diretto dalla domanda che scorre nella direzione opposta.

La programmazione dataflow non supporta né consente una soluzione accettabile al problema posto; però fornisce soluzioni parziali a due sotto problemi. Fornisce un modo per estendere esplicitamente ed anche dinamicamente l'algoritmo di produzione dei parametri attuali di una funzione e, un modo per definire un contesto per i parametri di una chiamata a funzione (che in generale fanno parte di diverse funzioni).

Conclusioni

Il paradigma dataflow è l'unico di quelli presentati che permette di programmare esplicitamente il flusso dei dati. Di conseguenza i dati in questo modello hanno un ruolo attivo, si propagano sul grafo dataflow verso tutte le operazioni che possono contribuire a calcolare. L'assenza del flusso della domanda impedisce di scrivere una funzione che provveda a farsi calcolare gli eventuali parametri mancanti.

Il paradigma dataflow contribuisce a risolvere alcuni sotto problemi dell'Obiettivo di Adattabilità. In particolare la copia della parte di grafo dataflow corrispondente ad una chiamata a funzione è un meccanismo per creare un contesto trasversale tra più funzioni che devono collaborare alla produzione di un risultato. Inoltre la composizione di funzioni forma una base di conoscenze riusabili.

Capitolo 18

PROGRAMMAZIONE LOGICA

La programmazione logica nasce dalla tradizione dei formalismi logici (sistemi deduttivi). I linguaggi di programmazione logica hanno trovato applicazioni soprattutto nel campo dell'intelligenza artificiale come ad esempio nei sistemi esperti basati sulla conoscenza.

Il programmatore dichiara gli obiettivi (asserzioni) e i fatti e le regole di inferenza per raggiungerli, ma non invoca esplicitamente queste regole per raggiungere gli obiettivi. È il motore inferenziale del sistema che cerca di raggiungere gli obiettivi applicando opportunamente le regole.

I linguaggi logici specificano il problema in uno stile dichiarativo, descrivono qual è l'obiettivo (che cosa deve fare il programma) e lasciano che sia il sistema a cercare di raggiungerlo.

Il modello di computazione logica

La macchina che fa da supporto all'esecuzione dei programmi logici è un interprete diretto dagli obiettivi; tale interprete viene chiamato *motore inferenziale*. L'operazione fondamentale svolta dal motore inferenziale è la *risoluzione* che è una ricerca esaustiva con backtracking che ha come fine trovare delle istanze di un obiettivo fissato che siano deducibili dal database di conoscenze definite nel programma. La ricerca avviene secondo un criterio determinato, partendo dall'obiettivo, in uno spazio di ricerca costituito dalla base di fatti e regole. Il successo o il fallimento di un obiettivo sono usati per controllare il flusso d'esecuzione del programma.

Anatomia della programmazione logica in Prolog

Il Prolog è un linguaggio di programmazione dichiarativo introdotto nel 1972 da Alain Colmerauer, Philippe Roussel e Robert Kowalsky [ColRou92]. Per un manuale di riferimento dello standard del linguaggio si rimanda a [DEC96].

Termini

In Prolog c'è un'unica struttura dati: i termini. I *termini* possono essere semplici o composti. Un termine semplice è una variabile oppure una costante numerica o un identificatore (atomo). Un termine composto è un atomo - detto funtore - seguito da una lista non vuota di termini tra parentesi. Gli atomi e i termini composti vengono chiamati *predicati* o *fatti*. Un termine composto il cui funtore principale sia il simbolo della dipendenza logica (:-) viene chiamato *regola.* Una regola definisce un predicato in termine di altri predicati. Regole e fatti vengono chiamati più in generale *clausole*.

Un termine composto ha l'aspetto di una chiamata a funzione e ne ha anche il comportamento quando viene posto come obiettivo (goal). Il Prolog non fa una distinzione esplicita tra parametri di ingresso e di uscita. Il sistema determina dinamicamente il ruolo dei parametri in base agli obiettivi dati.

Un *indicatore di predicato* è un termine della forma *nome/arità* ed è usato per denotare un predicato. La risoluzione procede per successive sostituzioni rese possibili dall'*unificazione* di un predicato dell'obiettivo con una clausola del database ed è limitata ai termini che hanno lo stesso indicatore di predicato.

Critica del modello logico

Il modello logico non richiede al programmatore di specificare la sequenza delle regole da applicare. Il motore inferenziale sceglie le regole dal database in base all'obiettivo che viene posto e ai vincoli di unificazione espressi dal programmatore nelle regole. L'ordine di valutazione viene indirettamente determinato dalla definizione delle dipendenze di un obiettivo dalle regole che immediatamente possono portare a raggiungerlo. Ed è sottoposto al vincolo che

un obiettivo non può essere raggiunto se prima non vengono raggiunti i sotto obiettivi definiti.

Mentre in un linguaggio imperativo i vincoli sequenziali legano una istruzione da eseguire alla successiva nella direzione che va dall'inizio della funzione alla sua fine; in un linguaggio logico i vincoli sequenziali legano una regola a quelle immediatamente precedenti nella direzione che va dal risultato che si vuole raggiungere ai fatti conosciuti. Dirò pertanto che in un programma imperativo i vincoli sequenziali esprimono una sequenza diretta mentre in un programma logico una sequenza inversa.

La sequenza inversa definita implicitamente dalle regole è in generale un albero con possibili cicli. Più regole alternative possono essere applicate per raggiungere uno stesso obiettivo. Il motore inferenziale esegue una ricerca esaustiva con backtracking sullo spazio del problema rappresentato da questo albero, alla ricerca di una soluzione. Per rappresentare adeguatamente un linguaggio di programmazione logico bisognerebbe ricorrere ad un ambiente di programmazione visuale che mostri l'albero diretto da un obiettivo in modo analogo a quanto fanno i linguaggi dataflow visuali per mostrare l'albero diretto dai dati.

In Prolog le clausole vengono definite singolarmente e anche nel database non formano un grafo orientato. Il motore inferenziale durante l'operazione di risoluzione costruisce dinamicamente il percorso del flusso della domanda con copie delle clausole che usa. Questa scelta equivale a creare un contesto trasversale tra l'obiettivo e i fatti che previene la possibilità di interferenze con altre risoluzioni di obiettivi.

Nella programmazione logica non esistono variabili e non si pone il problema della gestione delle celle di memoria; il programmatore può concentrarsi sulla definizione di fatti e regole. Il predicati predefiniti *assert* e *retract* del Prolog permettono rispettivamente di definire ed eliminare dei termini; possono

pertanto essere usati per ridefinire dei fatti e delle regole reintroducendo in questo modo il concetto di variabile con tanto di effetti collaterali.

Le clausole non vengono fatte circolare come avviene per i dati negli altri paradigmi di programmazione. In particolare, gli argomenti di un termine composto non devono essere passati come avviene per i parametri di una funzione imperativa o funzionale. Le clausole costituiscono il database ed è il motore inferenziale che, in base all'obiettivo da dimostrare va a prendere quelle unificabili. Come vedremo, al fine del raggiungimento dell'Obiettivo di Adattabilità questa è una differenza molto significativa.

Scelte arbitrarie

La scelte arbitrarie elencate di seguito non possono essere colte come tali ragionando nella tradizione dei linguaggi logici.

- È arbitrario fissare un inizio ed una fine propri di una clausola. La scelta è però ragionevole nell'ambito della programmazione logica. Per poter dimostrare un obiettivo, il processo di risoluzione procede eseguendo delle sostituzioni e per farlo ha bisogno che le parti che sostituisce siano delimitate. Rispetto alle funzioni imperative che contengono una sequenza di istruzioni, le clausole dei linguaggi logici hanno una granularità molto più fine che rende quasi trascurabili gli effetti di questa scelta arbitraria.

- È arbitrario stabilire che quando viene raggiunto un sotto obiettivo, la risoluzione proceda unicamente con l'obiettivo da cui è partita la ricerca. Il raggiungimento di un sotto obiettivo può utilmente essere portato avanti anche per altre dimostrazioni.

Sovraspecificazioni obbligate

I linguaggi logici obbligano il programmatore a sovraspecificare i programmi nel modo che segue. Per comprendere pienamente che si tratta di una sovraspecificazione è necessario porsi dal punto di vista di un'altra tradizione di

programmazione che non richieda di specificare questo aspetto di un programma.

- È necessario definire la struttura dei dati che modellano un'entità complessa. In Prolog questa sovraspecificazione si riflette principalmente nella scelta degli indicatori di predicato (nome/arità). Questa scelta limita l'unificazione e di conseguenza condiziona l'esito del processo di risoluzione.

Contributi all'Obiettivo di Adattabilità

Il paradigma logico è il solo a ripartire la responsabilità della determinazione dei parametri attuali però coinvolge tre soggetti: il chiamante, la funzione chiamata e la base di conoscenze. Il chiamante restringe il dominio della funzione di ricerca mentre la base di conoscenze contiene i parametri attuali e pertanto definisce il codominio della funzione di ricerca. La funzione chiamata determina un parametro attuale esplorando lo spazio delle soluzioni ristretto dagli altri due soggetti. In questo paragrafo oltre ad evidenziare i contributi positivi farò vedere che la parte di responsabilità sottratta dalla base di conoscenze limita l'espressività di questa soluzione.

Essendo diretto dalla domanda, un generico programma lavora sempre con richieste di strutture dati più o meno vincolate e non con riferimenti a particolari istanze. Quindi un parametro attuale viene determinato esplorando una molteplicità di candidati non uno solo prefissato. Questa soluzione rappresenta un superamento del problema del passaggio degli aggregati ed una alternativa ai puntatori. Le strutture dati restano ferme ma diversamente da quanto avviene usando i puntatori una richiesta vincolata produce una ricerca che lascia potenzialmente aperta la determinazione della struttura dati. Questa soluzione però non è sufficiente per superare il concetto di aggregato per il seguente motivo.

A causa della mancanza di un flusso dei dati, la ricerca diretta dall'obiettivo fatta dal motore inferenziale termina sempre su dati definiti strutturalmente cioè su

clausole che fanno parte della base di conoscenze e che pertanto esistono indipendentemente dal flusso di esecuzione del programma. Quindi o sono interessato a trovare tutte le soluzioni oppure devo scrivere l'algoritmo in modo che la prima trovata vada bene. Non posso scegliere algoritmicamente quale dato deve essere usato tra quelli compatibili con il flusso della domanda.

Il flusso dei dati avrebbe la responsabilità di attivare selettivamente alcune clausole in base alle necessità del contesto chiamante. La differenza fra attivare dei dati e passare dei dati è che nel primo caso fanno parte della base di conoscenza e quindi sono riusabili in altre chiamate mentre nel secondo caso fanno parte solo di una determinata chiamata. La differenza è apprezzabile soprattutto quando i parametri non sono semplici costanti ma espressioni.

L'unico meccanismo che si può provare a definire in Prolog per esprimere il flusso dei dati è basato sui predicati *assert* e *retract.* Purtroppo l'algoritmo di risoluzione standard del Prolog esegue la ricerca in profondità e senza preoccuparsi del problema della ricorsione infinita. Quindi anche se è definita una regola che permetterebbe di trovare il risultato, non è garantito che venga provata durante la ricerca. Di conseguenza il corretto funzionamento del meccanismo dipende dall'ordine di definizione delle regole ma, come si può vedere dall'esempio che segue, non esiste un ordine per evitare la ricorsione infinita. Per risolvere questo problema sarebbe sufficiente modificare l'algoritmo di risoluzione; il vero problema è che i parametri attuali forniti separatamente con delle *assert* non formano un contesto chiamante e pertanto possono interferire con altre chiamate.

```
% non funziona come passaggio di parametri
test1(X):-
    altezza(4),
    perimetro(18),
    area(X).

% funziona correttamente
test2(X):-
    asserta(altezza(4)),
    asserta(base(5)),
    area(X),
```

```
    retract(altezza(_)),
    retract(base(_)).

% funziona (a patto di non invertire l'ordine delle regole per
base/1)
test3(X):-
    asserta(area(20)),
    asserta(altezza(4)),
    base(X),
    retract(area(_)),
    retract(altezza(_)).

% non funziona se non scambio l'ordine delle regole per base/1
% e se lo faccio non va più test3/1
test4(X):-
    asserta(altezza(4)),
    asserta(perimetro(18)),
    area(X),
    retract(altezza(_)),
    retract(perimetro(_)).

% funziona quando non va test3/1 e viceversa
test5(X):-
    asserta(perimetro(20)),
    asserta(altezza(4)),
    base(X),
    retract(perimetro(_)),
    retract(altezza(_)).


% base di conoscenze iniziale

area(Area):-
    base(Base),altezza(Altezza),
    Area is Base*Altezza.

base(Base):-
    area(Area),altezza(Altezza),
    Base is Area/Altezza.

altezza(Altezza):-
    area(Area),base(Base),
    Altezza is Area/Base.

% regole aggiunte

perimetro(Perimetro):-
    base(Base),altezza(Altezza),
    Perimetro is (Base+Altezza)*2.

base(Base):-
    perimetro(Perimetro),altezza(Altezza),
    Base is (Perimetro/2)-Altezza.

altezza(Altezza):-
    perimetro(Perimetro),base(Base),
```

```
Altezza is (Perimetro/2)-Base.
```

La programmazione logica non supporta né consente una soluzione accettabile al problema posto; però fornisce una soluzione parziale a un sotto problema. Permette di estendere esplicitamente ed anche dinamicamente l'algoritmo di produzione dei parametri attuali di una funzione. Rispetto alla soluzione fornita dalla programmazione funzionale e da quella dataflow, questa ha il vantaggio di essere anche una soluzione al problema del passaggio dei parametri.

Un modello di esecuzione basato unicamente sul flusso della domanda porta a cercare tutti i modi conosciuti per produrre un particolare risultato; la ricerca in generale è molto più di quella che serve. Pertanto è opportuno fermare la ricerca su un ramo quando si raggiunge un dato e fermare l'intera ricerca quando viene prodotto il risultato in un qualche modo.

Abbiamo compreso che è necessario che un parametro attuale venga determinato esplorando una molteplicità di candidati non uno solo prefissato. Però non è sufficiente bisogna anche potere scegliere algoritmicamente quale dato deve essere usato tra quelli compatibili con il flusso della domanda.

Conclusioni

Nella programmazione dichiarativa l'esecuzione è diretta dal flusso della domanda; non c'è flusso dei dati. Di conseguenza il programmatore può limitarsi a dire *cosa* vuole ottenere e la richiesta si propaga verso tutti i sotto problemi che possono contribuire al raggiungimento del risultato. L'assenza di un flusso autonomo dei dati rende impossibile aggiungere parametri e passarli separatamente.

La definizione esplicita del flusso della domanda fa si che una nuova funzione si comporti come una estensione di quelle già definite. In Prolog questa funzionalità è fortemente ridimensionata dal fatto che il risultato è uno dei

parametri e non il valore di ritorno e dal fatto che l'unificazione è limitata ai termini che hanno lo stesso *indicatore di predicato* (nome/arità). La limitazione è cioè dovuta alla scelta di subordinare il flusso della domanda al meccanismo del pattern matching.

Capitolo 19

FONDAMENTI DI UNA NUOVA UNITÀ FUNZIONALE

Nell'analisi degli altri paradigmi di programmazione ho evidenziato le proprietà funzionali al raggiungimento dell'Obiettivo di Adattabilità; ma ho anche fatto vedere che non è possibile raggiungerlo in nessuno di essi. In questo capitolo mi propongo di definire in modo incrementale le caratteristiche che deve avere una nuova unità funzionale per supportare l'Obiettivo di Adattabilità. Lo scopo è far vedere che le condizioni necessarie trovate possono coesistere. Inoltre mi propongo di indicare come sviluppare la nuova unità funzionale per rispondere affermativamente a tutte le domande formulate nell'introduzione di questa parte della tesi.

Flussi di esecuzione

Considero un modello di esecuzione dei programmi basato sul flusso delle informazioni. Il *flusso di esecuzione* di un programma è costituito da due componenti: il flusso della domanda e il flusso dei dati.

Il *flusso della domanda* esprime il cosa voglio derivare; il *flusso dei dati* esprime il cosa è possibile derivare. Il *flusso del controllo* nasce dall'unione degli altri due flussi pertanto esprime il cosa voglio derivare e come voglio ottenerlo. Il flusso della domanda di un risultato denota un albero di propagazione che ha come radice il risultato da raggiungere e si estende verso tutte le operazioni che possono contribuire a produrlo sempre più indirettamente. Il flusso di un dato denota un albero di propagazione che ha come radice il dato disponibile e si estende verso tutte le operazioni che può contribuire a calcolare sempre più indirettamente. Si ha flusso del controllo quando una unità funzionale determina interamente il

percorso che a partire dalla domanda del risultato (che può essere anche la semplice esecuzione) arriva a produrlo.

I flussi nei paradigmi di programmazione

Nel paradigma imperativo, il flusso dei dati e il flusso della domanda sono completamente oscurati dal flusso del controllo nel senso letterale che non è possibile distinguere i due flussi componenti. I dati hanno esclusivamente un ruolo passivo: non si spostano mai verso quello che possono contribuire a calcolare; stanno fermi in attesa che un'istruzione li venga a prendere o modificare. Una chiamata a funzione, essendo anche il punto di ritorno, potrebbe essere considerata l'inizio di un flusso diretto dalla domanda (del risultato) che si propaga verso i dati che possono calcolarlo se non fosse che i parametri (dati) vengono passati tutti contestualmente alla chiamata a funzione rendendo non necessario oltre che impossibile un flusso autonomo diretto dalla domanda.

Anche l'esecuzione di un programma funzionale è dominata dal flusso del controllo. Però in questo paradigma sono distinguibili il flusso della domanda e il flusso dei dati. Il percorso dei due flussi è completamente determinato dalla definizione e dalla composizione delle funzioni però il motore inferenziale, grazie all'assenza di effetti collaterali, può scegliere la strategia di valutazione e privilegiare ad esempio il flusso della domanda (valutazione lazy). I dati possono avere un ruolo passivo e stare fermi in attesa che un'espressione li richieda oppure possono avere un ruolo attivo e propagarsi verso il risultato. In quest'ultimo caso si ha un flusso dei dati limitato al percorso scelto dal programmatore. Analogamente, una chiamata a funzione intesa come punto di ritorno può avere un ruolo passivo ed attendere che il flusso dei dati arrivi a produrre il risultato oppure può avere un ruolo attivo e propagare la richiesta verso i dati disponibili. In quest'ultimo caso si ha un flusso della domanda limitato al percorso scelto dal programmatore.

Nel paradigma dataflow viene definito solo il percorso del flusso dei dati. La richiesta di un risultato è passiva, consiste nell'attesa che il flusso dei dati arrivi a produrlo. I dati hanno un ruolo attivo e si propagano verso tutte le operazioni che possono contribuire ad eseguire. A titolo di ottimizzazione, si può attribuire un ruolo attivo alla richiesta di un risultato e propagarla verso i dati disponibili in modo da limitare il flusso dei dati alla produzione dei risultati richiesti. Questa strategia di esecuzione avvicina il paradigma dataflow a quello funzionale (lazy) non a quello logico perché il flusso della domanda che si introduce è limitato al percorso scelto dal programmatore per il flusso dei dati.

Nel paradigma logico viene definito solo il percorso del flusso della domanda. I dati hanno un ruolo passivo, attendono che il flusso della domanda venga a prenderli. La richiesta di un risultato ha un ruolo attivo e si propaga verso tutte le operazioni che possono contribuire a raggiungerlo finché non arriva ai dati; a quel punto inizia la produzione del risultato seguendo il percorso individuato.

Classificazione dei paradigmi di programmazione

Nella programmazione procedurale i percorsi dei due flussi di esecuzione sono completamente determinati. L'assenza di effetti collaterali permette di privilegiare il flusso della domanda rendendo la valutazione di un parametro attuale opzionale e segna il confine tra programmazione imperativa e funzionale. Infatti, un parametro non è un comando: "calcola questo!" ma è un suggerimento su come ottenere un valore se è necessario. Però il programmatore deve sempre specificare come produrre il valore e, se è necessario, quel valore viene calcolato con la funzione scelta e non provando una molteplicità di funzioni che possono calcolarlo. Questo segna il confine tra programmazione procedurale e programmazione dichiarativa. Il paradigma dataflow è procedurale perché definisce il percorso del flusso dei dati e non prevede un flusso della domanda. Inoltre è imperativo perché l'assenza di effetti collaterali non può rendere opzionale la valutazione dei parametri.

Nella programmazione procedurale il programmatore deve fare mentalmente una ricerca diretta dalla domanda, scegliere tutto l'albero di funzioni da usare per ottenere un valore e se rimangono dei parametri scoperti deve farseli passare. Il programmatore deve cioè sovraspecificare una chiamata a funzione: non può limitarsi a dire *cosa* vuole ottenere, deve preoccuparsi anche di scrivere *come* farlo. Questo pregiudica la possibilità di aggiungere ad un programma esistente nuovi modi per ottenere un risultato.

Step base: la nuova unità funzionale

Introduco una nuova unità funzionale che chiamo *Step* o *Passo* costituita nella sua versione base qui presentata da due linee di ingresso/uscita: *entry-call* e *done-action*. Associo ad ogni passo una rappresentazione visuale a forma di rombo; ogni lato del rombo è sede di un ingresso/uscita come riportato in Figura 1.

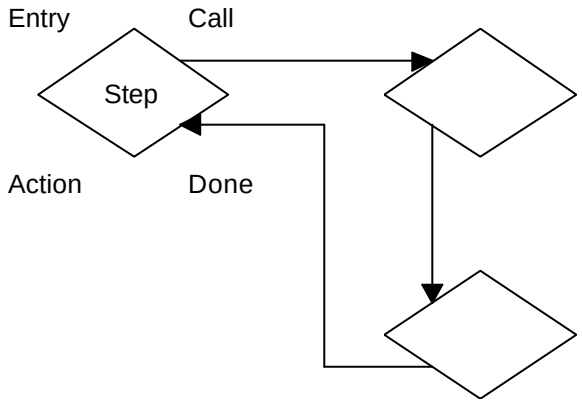

Figura 1 Step base: ingressi e uscite

Gli ingressi e le uscite sono collegati da *Link* o *Connessioni* orientate nella direzione che va dall'uscita di un passo all'ingresso di un altro; ad ogni uscita possono essere collegate un numero a piacere di connessioni uscenti e analogamente in ogni ingresso possono confluire un numero a piacere di connessioni entranti. Il comportamento di un passo è definito dal diagramma degli stati rappresentato in Figura 2.

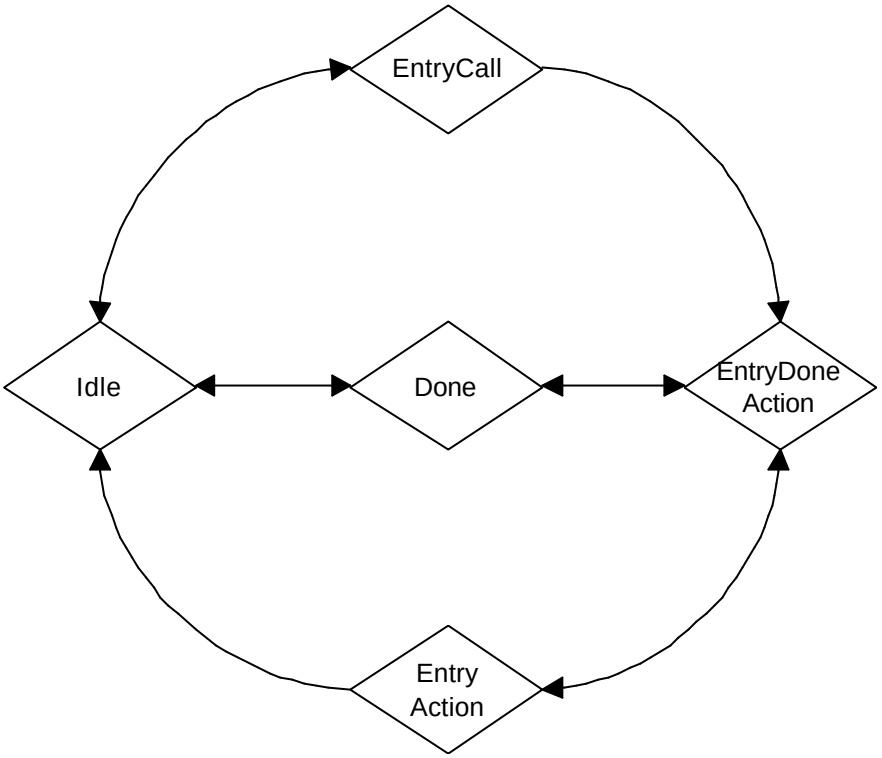

Figura 2 Step base: diagramma degli stati

L'attivazione di un ingresso è sostenuta; pertanto è accompagnata da un cambiamento di stato sia all'inizio che alla fine dell'attivazione. Dal diagramma degli stati si possono notare tre interessanti proprietà. Una richiesta (entry) si propaga (call) solo se il risultato (done) non è disponibile. Un risultato (done) si propaga (action) solo quando viene richiesto (entry). Infine, la propagazione del risultato (action) è una continuazione della richiesta (entry) e non dipende dalla persistenza del risultato (done).

Lungo l'asse *entry-call* si possono definire *sequenze inverse* dal passo che produce il risultato verso quelli che ricevono i parametri attuali (vedi Figura 3). La propagazione della richiesta prosegue fintanto che non viene raggiunto un dato (done) disponibile. La propagazione avviene in parallelo verso tutti i modi per completare un passo; il primo che ce la fa interrompe gli altri tentativi. La sequenza inversa *entry-call* è l'unico percorso che permette una propagazione autonoma del flusso. Ogni passo può essere interpretato come un ritorno (done)

della funzione iniziata al passo precedente unito ad una chiamata a funzione (call) per raggiungere il passo successivo. L'unico modo che hanno le funzioni chiamate da un passo per influenzare l'esecuzione delle funzioni chiamate al passo successivo è comunicando tramite una funzione in esecuzione concorrente ad entrambi i passi.

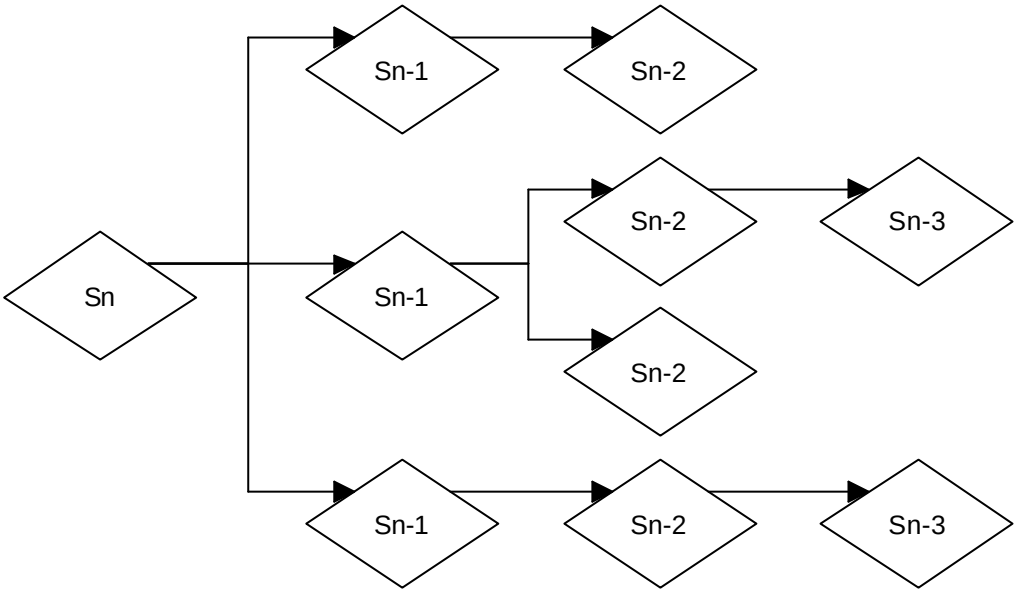

Figura 3 Flusso della domanda – sequenza inversa

Lungo l'asse *entry-action* si possono definire sequenze dirette, ogni passo è connesso con i passi successivi che rappresentano delle continuazioni (vedi Figura 4). Il flusso procede un passo alla volta: propaga il flusso della domanda di un passo e attende la sua terminazione (done) prima di iniziare il successivo. Ogni passo può essere interpretato come una chiamata a delle funzioni parallele (call) con ritorno (done). L'unico modo che hanno le funzioni chiamate da un passo per influenzare l'esecuzione dei passi successivi è comunicando tramite una funzione in esecuzione concorrente ad entrambi i passi. Più in generale, le funzioni non possono produrre effetti collaterali, possono solo comunicare con funzioni concorrenti.

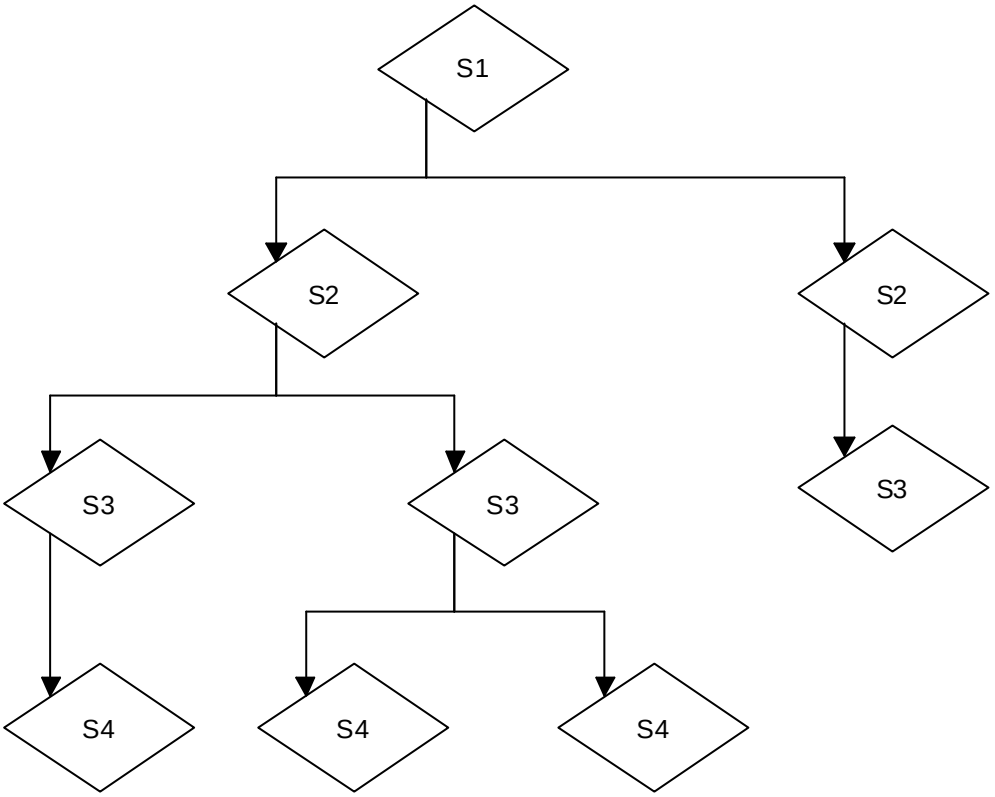

Figura 4 Flusso della domanda - sequenza diretta

Lungo l'asse *entry-call*, nella direzione opposta, si possono definire anche *sequenze dirette* dal passo che riceve un dato verso tutte le continuazioni che possono richiederlo (vedi Figura 5). Il flusso dei dati procede un passo alla volta: attende la richiesta prima di proseguire.

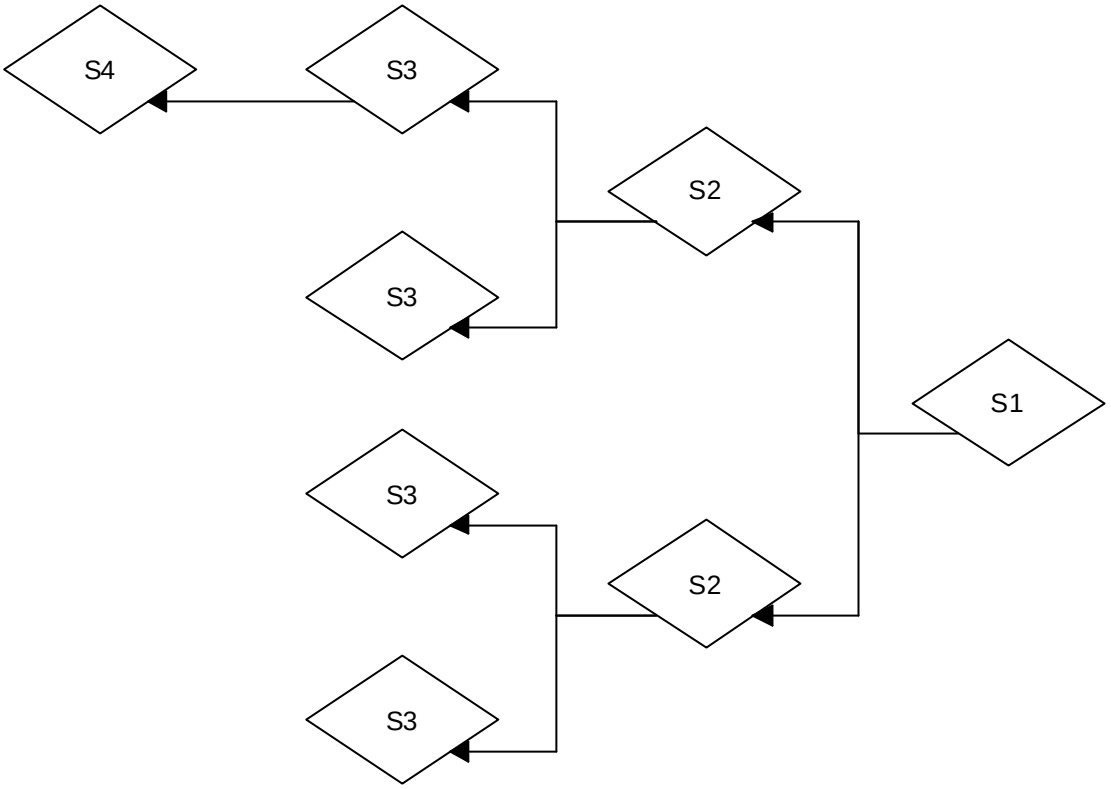

Figura 5 Flusso dei dati subordinato alla domanda

Condizioni per una soluzione all'Obiettivo di Adattabilità

La domanda fondamentale che ci ha accompagnati per tutta questa parte della tesi è se sia possibile ripartire la responsabilità della determinazione dei parametri attuali tra la chiamata a funzione e la funzione chiamata.

La nuova unità funzionale è definita in modo che ogni passo di esecuzione se non può procedere con action perché manca done inizia la propagazione (call) della richiesta di done in modo da determinare, attivandolo in modo sostenuto, un percorso per produrre done a partire dai dati disponibili nel contesto chiamante. Ogni volta che la propagazione della richiesta raggiunge un dato

inizia un flusso dei dati (mostrato in Figura 6) limitato al percorso che porta al risultato.

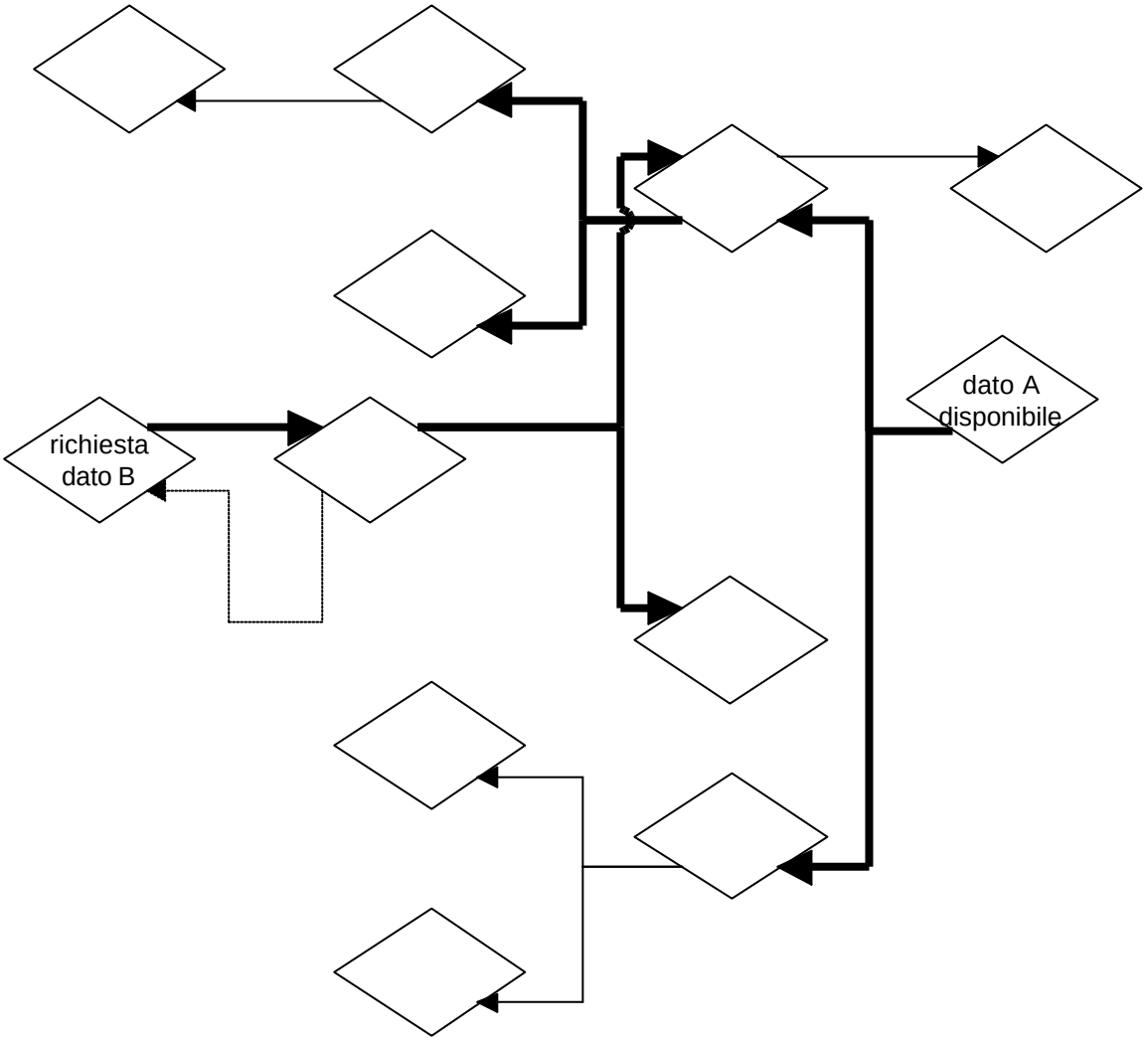

Figura 6 Produzione su richiesta di un dato

Nel caso specifico oggetto della domanda, se un parametro attuale non viene fornito dal chiamante sulla done del passo che rappresenta il corrispondente parametro formale, quest'ultimo provvede a chiamare un albero di funzioni composte che lo calcola a partire dai dati disponibili nel contesto chiamante. In alternativa, il chiamante può passare direttamente un dato disponibile oppure può chiamare un determinato insieme di funzioni per calcolare un valore da

passare o infine può restringere le funzioni usabili dalla funzione chiamata per calcolare un parametro attuale.

Vediamo ora più in dettaglio se questa soluzione è sufficiente a risolvere tutti gli altri problemi posti.

Posso scrivere una funzione che possa essere chiamata con un sottoinsieme eventualmente vuoto dei parametri di cui ha bisogno per produrre il risultato e che provveda a farsi calcolare quelli mancanti?

Tutte le funzioni si comportano come richiesto senza bisogno di definirle in modo particolare. Nel modello proposto non esistono funzioni in quanto tali ma solo rispetto al contesto chiamante pertanto non è distinguibile l'inizio della funzione né i punti in cui vengono passati i parametri attuali. Ogni passo di esecuzione è legato ai passi che possono precederlo; se viene passato un dato, quello fissa l'inizio della funzione per il parametro scelto altrimenti la funzione inizierà più a monte da altri dati disponibili nel contesto chiamante.

Posso estendere esplicitamente e dinamicamente l'algoritmo di ricerca/produzione dei parametri attuali?

Se i due contesti – chiamante e chiamato – sono separati il programmatore è tenuto a definire i segmenti di funzione mancanti e può scegliere come ripartire la responsabilità del calcolo. In generale, poiché ogni passo definisce un tratto dei percorsi dei flussi della domanda e dei dati, la normale attività di programmazione consiste nell'estendere l'algoritmo di ricerca/produzione dei parametri attuali. Affinché l'estensione possa avvenire dinamicamente è necessario che le operazioni di programmazione siano accessibili in una qualche forma strutturata al programma.

Posso fare in modo che i parametri attuali determinati separatamente dalla funzione chiamata formino un unico contesto chiamante; in particolare non interferiscano con altre chiamate?

No. Il modello così come è stato definito finora non è sufficiente ad evitare fenomeni di interferenza tra chiamate a funzione. Si deve aggiungere un generatore di contesti analogo a quelli presenti in tutti gli altri paradigmi.

Posso trovare un sostituto ai puntatori che lasci aperta la determinazione dell'oggetto puntato?

Il modello base mi permette di esplicitare le componenti dell'operazione di dereferenziazione e distinguere il flusso della domanda dal flusso dei dati. Gli algoritmi possono essere definiti esplicitando solo la parte della domanda a meno che non serva fissare anche alcuni dati. In generale le funzioni possono essere definite senza che si assumano la responsabilità di determinare i parametri attuali.

Posso rappresentare un oggetto in forma disaggregata ed usare il sostituto del puntatore per raggiungere ugualmente tutti gli attributi di un oggetto come se fosse ancora unitario?

Il concetto di oggetto inteso come aggregato di attributi è fortemente legato al paradigma imperativo e all'architettura della memoria di un calcolatore tradizionale. Un oggetto viene mappato su un'area di memoria contigua, in questo modo la funzione di instradamento incentrata sull'oggetto risulta particolarmente semplice: con un puntatore alla base dell'oggetto la richiesta di accedere ad un campo si risolve aggiungendo alla base il relativo scostamento.

Per rappresentare un oggetto in forma disaggregata bisogna enfatizzare il concetto di identità come legame per unire gli attributi. Inoltre la funzione di instradamento deve essere incentrata sugli attributi.

Quindi devo definire un passo che rappresenta l'identità di un oggetto ed è connesso (ad esempio su done) a tutti i passi che rappresentano gli attributi. Per scrivere un passo che accede ad un attributo devo collegarlo (ad esempio su entry) a tutti i passi che rappresentano le occorrenze dell'attributo. Il risultato è

che quando il passo richiede l'attributo l'unico attivo su done è quello dell'oggetto a cui si vuole accedere.

Posso come programmatore sottrarmi alla responsabilità di definire la struttura per rappresentare una entità complessa?

Non ho bisogno di pormi il problema di rappresentare una entità complessa. Scrivo gli algoritmi che usano le entità e i loro attributi inserendo al posto di una entità una chiamata alle funzioni che la determinano e al posto di un attributo una chiamata alle funzioni che lo determinano. Il risultato è una rappresentazione disaggregata delle entità. Per ogni attributo ho un albero di funzioni che determinano in base al contesto chiamante una occorrenza dell'attributo. Per ogni entità ho un albero di funzioni aggreganti che lega tutti gli attributi. Quando voglio accedere ad un attributo mi devo porre il problema di specificare il contesto della richiesta.

Sulla necessità di avere flussi separati

Nessun paradigma attuale supporta una programmazione esplicita dei percorsi che devono essere seguiti dai due flussi di esecuzione. Nel paradigma imperativo i due flussi sono sempre accoppiati e si può parlare solo di flusso del controllo. Nel paradigma funzionale viene definito un unico percorso del controllo ma su di esso si può scegliere una strategia di esecuzione che esplicita i due flussi. Il paradigma dataflow definisce solo il percorso del flusso dei dati. Infine il paradigma logico definisce solo il percorso del flusso della domanda.

Un modello di esecuzione basato unicamente sul flusso dei dati porta a calcolare tutto ciò che è derivabile; si calcola in generale molto di più di quello che viene richiesto dal programma. Pertanto è opportuno limitare l'avanzamento dei dati all'effettiva richiesta.

Un modello di esecuzione basato unicamente sul flusso della domanda porta a cercare tutti i modi conosciuti per produrre un particolare risultato; si cerca in generale molto più di quello che serve. Pertanto è opportuno fermare la ricerca

su un ramo quando si raggiunge un dato e fermare l'intera ricerca quando viene prodotto il risultato in un qualche modo.

Le soluzioni proposte richiedono che un parametro attuale venga determinato esplorando una molteplicità di candidati non uno solo prefissato e bisogna anche potere scegliere algoritmicamente quale dato deve essere usato tra quelli compatibili con il flusso della domanda.

Sulla necessità di aggiungere un generatore di contesti

Le chiamate a funzione nel modello base definito possono interferire nei seguenti modi. I parametri attuali e i dati disponibili formano un unico contesto comune a tutto il programma e non contesti separati per ogni chiamata che lo richieda. Di conseguenza una funzione può usare parametri attuali provenienti da diverse chiamate. Inoltre, il risultato di una chiamata a funzione può essere ricevuto anche da altre chiamate con diversi parametri attuali.

Per risolvere questi problemi è necessario aggiungere al modello base un meccanismo per generare contesti. Analogamente agli altri paradigmi la soluzione può essere una combinazione di copia ed esecuzione in mutua esclusione.

La sequenza inversa definisce tutti i modi per raggiungere un certo risultato. Quando una chiamata a funzione sceglie inizio e fine della funzione chiamata è inutile che il flusso della domanda esplori altre possibilità se esistono vuole dire che si è verificata una interferenza. Pertanto posso fare una copia della sequenza determinata ed escludere la possibilità di interferenze.

Il generatore di contesti, più in generale, può essere sviluppato come parte del meccanismo di allocazione strutturale pure necessario. Diversamente dalle operazioni *new/clone* fornite dagli attuali linguaggi, è qui necessario fare in modo che i passi connessi alla struttura originale possano scegliere se diventare connessi anche alla copia allocata. Deve cioè essere strutturalmente definibile la

profondità della copia. Diverse soluzioni presentate richiedono questa possibilità.

Capitolo 20

CONCLUSIONI

I linguaggi attuali obbligano il programmatore a sovraspecificare i programmi con delle scelte vincolanti per l'utente finale, ma che non riguardano gli algoritmi.

Tra i costrutti linguistici finora ritenuti estranei al problema della sovraspecificazione vi sono tutti gli identificatori (variabili o immutabili) legati a valori: parametri di funzioni, puntatori, riferimenti, campi di oggetti.

L'accesso ad un identificatore si realizza con un flusso diretto dal controllo perché è insieme una domanda e una risposta (il valore) vincolati. Un puntatore ad un oggetto rappresenta ancora un flusso diretto dal controllo ma è possibile definire il comportamento dell'oggetto in modo che separi i due flussi. Le implementazioni dell'operazione di differenziazione trovate funzionano per questo motivo.

L'esistenza di un legame diretto tra identificatore e valore mi obbliga ad assumermi la responsabilità di assegnare un valore agli identificatori che uso oppure di farmelo passare. In questo secondo caso mi assumo la responsabilità di definire un percorso di inizializzazione. Non ho la terza possibilità di delegare la responsabilità dell'inizializzazione al primo chiamante che ha la necessità algoritmica di assegnare un valore oppure considerarlo un canale di comunicazione sincrono che attende un messaggio da una funzione concorrente.

La necessità di definire la struttura dei dati che modellano un'entità complessa è da attribuire a questa forma di sovraspecificazione. Il cliente di un attributo di un oggetto deve conoscere e quindi aver definito la classe che contiene

l'attributo, e la struttura del percorso dall'oggetto cliente all'oggetto dell'attributo.

Quando in una funzione ne chiamo una seconda lo faccio solo perché sono interessato al risultato che calcola eppure sono tenuto a procurarmi (calcolare o farmi passare) tutti i parametri richiesti dalla funzione chiamata anche se non sono funzionali al mio algoritmo e non avrei bisogno di vincolarli. Il programmatore non può limitarsi a dire *cosa* vuole ottenere, deve preoccuparsi anche di scrivere *come* farlo. Questo pregiudica la possibilità di aggiungere ad un programma esistente nuovi modi per ottenere un risultato.

La nuova unità funzionale presentata si distingue proprio per la possibilità che offre di non sovraspecificare i programmi. Per ottenere questo risultato è necessario poter programmare separatamente il flusso della domanda e il flusso dei dati.

Le funzioni e in generale gli identificatori, salvo diversa necessità, devono essere definiti lungo la sequenza inversa (del flusso della domanda). In questo modo, ad esempio, chiamante e chiamato sono legati da una comune richiesta non da un comune valore. Il programmatore è responsabile di definire (in modo dichiarativo) delle funzioni per determinare il valore di un identificatore nei diversi contesti; così facendo, implicitamente, definisce una entità in forma disaggregata e sottrae ai clienti di un identificatore la responsabilità di definire esplicitamente una struttura per le entità e un percorso per ottenere gli attributi.

BIBLIOGRAFIA

[AghKim98] Gul A. Agha, Wooyoung Kim, *Actors: a Unifying Model for Parallel and Distributed Computing*, 1998.

[AMST93] Gul A. Agha, Ian A. Mason, Scott F. Smith, Carolyn L. Talcott, *A Foundation for Actor Computation*, Journal of Functional Programming, 1(1), Cambridge University Press, 1993

[ArnGos96] Ken Arnold, James Gosling, *The Java Programming Language*, Massachusetts, Addison-Wesley, 1996.

[BMRSS96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerland, Michael Stal, *Pattern-oriented software architecture – A system of patterns*, England, Wiley, 1996.

[Cha93] Craig Chambers, *Predicate Classes*, ECOOP '93, 1993.

[ColRou92] Alain Colmerauer, Philippe Roussel, *The birth of Prolog*, PrologIA, 1992.

[DEC96] P.Deransart, A.Ed-Dbali, L.Cervoni, *Prolog: The Standard*, Springer-Verlag, 1996.

[EKC98] Michael Ernst, Craig Kaplan, Craig Chambers, *Predicate Dispatching: A Unified Theory of Dispatch*, ECOOP '98, 1998, 186-211.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns – Elements of Reusable Object-Oriented Software*, Massachusetts, Addison-Wesley, 1995.

[GJS96] James Gosling, Bill Joy, Guy Steele, *The Java Language Specification*, http://java.sun.com/docs/books/jls/html/index.html, Massachusetts, Addison-Wesley, 1996.

[Hew73] C.E.Hewitt, P.Bishop, R.Steiger, *A Universal Modular ACTOR Formalism for Artificial Intelligence*, Proceedings of the 3rd IJCAI, Stanford, California, 1973, 235-245.

[Jag95] R.Jagannathan, *Dataflow Models*, Computer Science Laboratory SRI International, California, 1995.

[KLMM97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina V. Lopes, Jean-Marc Loingtier, John Irwin, *Aspect-Oriented Programming*, Springer-Verlag, 1997.

[KOHK96] Matthew Kaplan, Harold Ossher, William Harrison, Vincent Kruskal, *Subject-oriented design and the Watson Subject Compiler*, OOPSLA '96 Subjectivity Workshop Position Paper, 1996.

[Lie86] Henry Lieberman, *Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems*, Proceedings of First ACM Conference on Object-Oriented Programming Systems, Languages and Applications, Portland, 1986.

[LinYel96] Tim Lindholm, Frank Yellin, *The Java Virtual Machine Specification*, http://java.sun.com/docs/books/vmspec/html/VMSpecTOC.doc.html, Massachusetts, Addison-Wesley, 1996.

[MRB98] Robert Martin, Dirk Riehle, Frank Buschmann, *Pattern Languages of Program Design 3*, Massachusetts, Addison-Wesley, 1998

[OHBS94] Harold Ossher, William Harrison, Frank Budinsky, Ian Simmonds, *Subject-Oriented Programming: Supporting Decentralized Development of Objects*, Proceedings of the 7th IBM Conference on Object-Oriented Technology, 1994.

[Opd92] William F. Opdyke, *Refactoring object-oriented frameworks*, PhD thesis, University of Illinois, Urbana-Champaign, 1992.

[PeyHug99] Simon Peyton Jones, John Hughes editors, *Haskell 98 – A non-strict, Purely Functional Language*, Report on the Programming Language, 1999.

[PeyLes92] Simon Peyton Jones, David Lester, *Implementing Functional Languages – A Tutorial*, Prentice Hall, 1992.

[SLU88] Lynn Andrea Stein, Henry Lieberman, David Ungar, *A Shared View of Sharing: The Treaty of Orlando*. In *Object-oriented concepts, applications, and databases*, Kim W., Lochowsky F., eds, Addison-Wesley, 1988, 31-48.

[Smi94] Randall B. Smith, *Prototype-Based Languages: Object Lessons from Class-Free Programming*, Sun Microsystems Laboratories, Mountain View, California, 1994.

[Sne93] David F.Snelling, *The Design and Analysis of a Stateless Data-Flow Architecture*, PhD Thesis, University of Manchester, 1993.

[SneEga94] David F.Snelling, Gregory K.Egan, *A Comparative Study of Data-Flow Architectures*, Technical Report Number UMCS-94-4-3, University of Manchester, 1994.

[Sol99a] Riccardo Solmi, *Meccanismi di condivisione – ereditarietà, delega, concatenazione*, Università di Bologna, 1999.

[Str91] Bjarne Stroustrup, *The C++ Programming Language*, 2nd ed., Massachusetts, Addison-Wesley, 1991.

[Sun99] Sun, *Java 2Platform API Specification*, http://java.sun.com/products/jdk/1.2/docs/api/index.html, Sun microsystems, 1999

[Sun99a] Sun, *The Java Tutorial*, http://java.sun.com/docs/books/tutorial/index.html, Sun microsystems, 1999

[Tai92] Antero Taivalsaari, *Kevo – a prototype-based object-oriented language based on concatenation and module operations*, TR DCS-197-1R, University of Victoria, 1992.

[Tai93] Antero Taivalsaari, *A critical view of inheritance and reusability in object-oriented programming*, PhD dissertation, University of Jyvaskyla, Finland, 1993.

[Tan92] Andrew S. Tanenbaum, *Modern operating systems*, Prentice-Hall International, Inc, 1992.

[UCCH91] David Ungar, Craig Chambers, Bay-Wei Chang, Urs Holzle, *Organizing Programs Without Classes*, Lisp and Symbolic Computation, 4(3), 1991, 223-24

[WinFlo87] Terry Winograd, Fernando Flores, *Calcolatori e conoscenza: un nuovo approccio alla progettazione delle tecnologie dell'informazione*, Milano, Mondadori, 1987.

[Xer99] Xerox, *AspectJ – Cross-cutting Objects for Better Modularity*, Official home site, http://www.aspectj.org, Xerox Corporation, 1999.