Appunti per le lezioni di Ingegneria del Software

corso di laurea in Ingegneria informatica
Università di Pisa

Andrea Domenici

A.A. 2021–2022

In memoria di
Anna Maria Gherardi e Franco Domenici
con gratitudine

Et sunt item qui scire volunt ut scientiam suam vendant; verbi causa, pro pecunia, pro honoribus: et turpis quæstus est. Sed sunt quoque qui scire volunt, ut ædificent: et charitas est. Et item qui scire volunt, ut ædificentur: et prudentia est.

— Bernardus Claræ vallensis, *Sermones in Cantica Canticorum*, SERMO XXXVI.

Indice

Capitolo 1

Introduzione

L'ingegneria del software non mi piace. Mi piacciono le cose pratiche, mi piace programmare in assembler.
– confessione autentica di uno studente

Questa dispensa riassume i contenuti dell'insegnamento di Ingegneria del software per il corso di laurea triennale in Ingegneria informatica. Tutte le nozioni qui esposte, incluse le parti non trattate a lezione, saranno materia di esame, escluse le appendici. Saranno materia di esame anche le nozioni presentate nelle ore di laboratorio. Il materiale raccolto nella dispensa dovrà essere integrato con i libri di testo indicati dal docente. Le appendici raccolgono materiale **non aggiornato** non piú in programma.

La pagina web del corso si trova su
`http://www2.ing.unipi.it/~a009435/issw/isw.html`.

1.1 Prime definizioni

L'ingegneria del software è l'insieme delle teorie, dei metodi e delle tecniche che si usano nello sviluppo industriale del software. Possiamo iniziarne lo studio considerando le seguenti definizioni:

> *"L'ingegneria del software è il settore dell'informatica che si occupa della creazione di sistemi software talmente grandi o complessi da dover essere realizzati da piú squadre di ingegneri. Di solito questi sistemi esistono in varie versioni e rimangono in servizio per parecchi anni. Durante la loro vita subiscono numerose modifiche [...]".*
> — C. Ghezzi *et al.*, Ingegneria del software: fondamenti e princípi [18]

> *"disciplina tecnologica e manageriale che riguarda la produzione sistematica e la manutenzione dei prodotti software, [...] sviluppati e modificati entro i tempi e i costi preventivati".*
> — D. Fairley, Software Engineering Concepts [13]

I concetti chiave di queste definizioni sono la *complessità* dei sistemi software, la loro *evoluzione* nel tempo, la necessità di lavoro *collettivo* che comporta la sua *gestione*, la *sistematicità* del processo di produzione, ed il rispetto di *vincoli economici*.

1.2 Concetti generali

In questo corso ci limiteremo agli aspetti piú tradizionalmente ingegneristici della materia, e purtroppo (o forse per fortuna) sarà possibile trattare solo pochi argomenti. Questi dovrebbero però dare una base sufficiente ad impostare il lavoro di sviluppo del software nella vita professionale. Il corso si propone di fornire le nozioni fondamentali sia su argomenti di immediata utilità pratica, sia su temi rilevanti per la formazione culturale dell'ingegnere del software.

In questa sezione vogliamo introdurre alcuni motivi conduttori che ricorreranno negli argomenti trattati in queste dispense.

1.2.1 Le parti in causa

Lo sviluppo e l'uso di un sistema software coinvolge molte persone, ed è necessario tener conto dei loro ruoli, delle esigenze e dei rapporti reciproci, per ottenere un prodotto che risponda pienamente alle aspettative. Per questo l'ingegneria del software studia anche gli aspetti sociali ed organizzativi sia dell'ambiente in cui viene sviluppato il software, sia di quello in cui il software viene applicato.

Questo aspetto dell'ingegneria del software non verrà trattato nel nostro corso, ma qui vogliamo chiarire alcuni termini che saranno usati per designare alcuni ruoli particolarmente importanti:

sviluppatore: sono sviluppatori[1] coloro che partecipano direttamente allo sviluppo del software, come *analisti*, *progettisti*, *programmatori*, o *collaudatori*. Due o piú ruoli possono essere ricoperti da una stessa persona. In alcuni casi il termine "sviluppatore" verrà contrapposto a "collaudatore", anche se i collaudatori partecipano al processo di sviluppo e sono quindi sviluppatori anch'essi.

produttore: per "produttore" del software si intende un'organizzazione che produce software, o una persona che la rappresenta. Uno sviluppatore generalmente è un dipendente del produttore.

committente: un'organizzazione o persona che chiede al produttore di fornire del software.

utente: una persona che usa il software. Generalmente il committente è distinto dagli utenti, ma spesso si userà il termine "utenti" per riferirsi sia agli utenti propriamente detti che al committente, ove non sia necessario fare questa distinzione.

[1] In questa dispensa il genere *grammaticale* maschile si riferisce a persone, animali o cose di qualsiasi natura, convinzione, propensione o tendenza di carattere sessuale, ideologico, culturale, religioso, artistico, letterario etc.

Osserviamo che l'utente di un software può essere a sua volta uno sviluppatore, nel caso che il software in questione sia un ambiente di sviluppo, o una libreria, o qualsiasi altro strumento di programmazione.

Osserviamo anche che spesso il software non viene sviluppato per un committente particolare (applicazioni *dedicate*, o *custom*, *bespoke*), ma viene messo in vendita (o anche distribuito liberamente) come un prodotto di consumo (applicazioni *generiche*, o *shrink-wrapped*).

1.2.2 Specifica e implementazione

Una *specifica* è una descrizione precisa dei *requisiti* (proprietà o comportamenti richiesti) di un sistema o di una sua parte. Una specifica descrive una certa entità "dall'esterno", cioè dice quali servizi devono essere forniti o quali proprietà devono essere esibite da tale entità. Per esempio, la specifica di un palazzo di case popolari potrebbe descrivere la capienza dell'edificio dicendo che deve poter ospitare cinquanta famiglie. Inoltre la specifica può indicarne il massimo costo ammissibile.

Il grado di precisione richiesto per una specifica dipende in generale dallo *scopo* della specifica. Dire che uno stabile deve ospitare cinquanta famiglie può essere sufficiente in una fase iniziale di pianificazione urbanistica, in cui il numero di famiglie da alloggiare è effettivamente il requisito piú significativo dal punto di vista del committente (il Comune). Questo dato, però, non è abbastanza preciso per chi dovrà calcolare il costo e per chi dovrà progettare lo stabile. Allora, usando delle tabelle standard, si può esprimere questo requisito come volume abitabile. Questo dato è correlato direttamente alle dimensioni fisiche dello stabile, e quindi diviene un parametro di progetto. Possiamo considerare il numero di famiglie come un *requisito dell'utente* (nel senso che questa proprietà è richiesta *dall*'utente, o piú precisamente, in questo caso, dal committente) e la cubatura come un *requisito del sistema* (cioè una proprietà richiesta *al* sistema). Anche nell'industria del software è necessario, in generale, produrre delle specifiche con diversi livelli di dettaglio e di formalità, a seconda dell'uso e delle persone a cui sono destinate.

Una specifica descrive *che cosa* deve fare un sistema, o quali proprietà deve avere, ma non *come* deve essere costruito per esibire quel comportamento o quelle proprietà. La specifica di un palazzo non dice come è fatta la sua struttura, di quanti piloni e travi è composto, di quali dimensioni, e via dicendo, non dice cioè qual è la *realizzazione* di un palazzo che soddisfi i requisiti specificati (nel campo del software la realizzazione di solito si chiama *implementazione*).

Un palazzo è la realizzazione di una specifica, ma naturalmente la costruzione del palazzo deve essere preceduta da un progetto. Un *progetto* è un insieme di documenti che descrivono *come* deve essere realizzato un sistema. Per esempio, il progetto di un palazzo dirà che in un certo punto ci deve essere una trave di una certa forma con certe dimensioni. Questa descrizione della trave è, a sua volta, una specifica dei requisiti (proprietà fisiche richieste) della trave stessa. Infine, la trave "vera", fatta di cemento, è la realizzazione di tale specifica. Possiamo quindi vedere il "processo di sviluppo" di un palazzo come una serie di passaggi: si produce una prima specifica orientata alle esigenze

del committente, poi una specifica orientata ai requisiti del sistema, poi un progetto che da una parte è un'implementazione delle specifiche, e dall'altra è esso stesso una specifica per il costruttore.

Dal punto di vista del progettista, un requisito è un obbligo imposto dall'esterno (e quindi fa parte della specifica) mentre l'implementazione è il risultato di una serie di scelte. Certe caratteristiche del sistema che potrebbero essere scelte di progetto, in determinati casi possono diventare dei requisiti: per esempio, i regolamenti urbanistici di un comune potrebbero porre un limite all'altezza degli edifici, oppure le norme di sicurezza possono imporre determinate soluzioni tecniche. Queste caratteristiche, quindi, non sono piú scelte dal progettista ma sono *vincoli* esterni. Piú in generale, un vincolo è una condizione che il sistema deve soddisfare, imposta da esigenze ambientali di varia natura (fisica, economica, legale...) o da limiti della tecnologia, che rappresenta un limite per le esigenze dell'utente o per le scelte del progettista. Un vincolo è quindi un requisito indipendente dalla volontà dell'utente.

Data la potenziale ambiguità nel ruolo (requisito o vincolo o scelta di progetto) di certi aspetti di un sistema, è importante che la documentazione prodotta durante il suo sviluppo identifichi tali ruoli chiaramente.

1.2.3 Modelli e linguaggi

Per quanto esposto nella sezione precedente, il processo di sviluppo del software si può vedere come la costruzione di una serie di *modelli*. Un modello è una descrizione astratta di un sistema, che serve a studiarlo prendendone in considerazione soltanto quelle caratteristiche che sono necessarie al conseguimento di un certo scopo. Ogni sistema, quindi, dovrà essere rappresentato per mezzo di piú modelli, ciascuno dei quali ne mostra solo alcuni aspetti, spesso a diversi livelli di dettaglio.

Ovviamente un modello deve essere espresso in qualche linguaggio. Una parte considerevole di questo corso verrà dedicata a linguaggi concepiti per descrivere sistemi, sistemi software in particolare.

Un linguaggio ci offre un *vocabolario* di simboli (parole o segni grafici) che rappresentano i concetti necessari a descrivere certi aspetti di un sistema, una *sintassi* che stabilisce in quali modi si possono costruire delle espressioni (anche espressioni grafiche, cioè diagrammi), e una *semantica* che definisce il significato delle espressioni.

Nel séguito si userà spesso il termine "*formalismo*", per riferirsi ad una famiglia di linguaggi che esprimono concetti simili (avendo quindi un modello teorico comune) e hanno sintassi simili (e quindi uno stile di rappresentazione comune). Questi linguaggi peraltro possono presentare varie differenze concettuali o sintattiche.

Letture
Obbligatorie: cap. 1 Ghezzi, Jazayeri, Mandrioli [18], oppure sez. 1.1–1.2 Ghezzi et al., oppure cap. 1 Pressman [40].

Capitolo 2

Il software e la sicurezza

Prima di entrare nella parte centrale del corso, questo capitolo presenta due casi molto noti di malfunzionamenti del software, cercando di illustrare la complessità dei fattori che li hanno provocati.

2.1 Il caso Therac-25

Il Therac-25 (fig. 2.1[1]) era una macchina per radioterapia che provocò almeno sei gravi incidenti di sovradosaggio, di cui tre mortali, fra il 1985 e il 1987 [25][2].

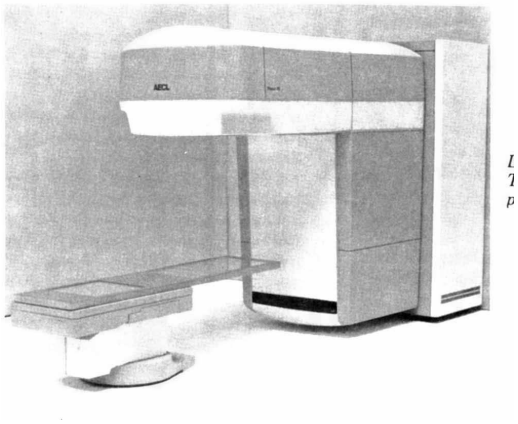


Figura 2.1: Il Therac-25.

Questa macchina generava un fascio di elettroni che potevano irradiare il paziente direttamente (terapia ad elettroni) o colpire un bersaglio metallico, producendo un fascio di raggi X (terapia a raggi X). All'uscita del fascio elettronico, un vassoio portaaccessori girevole permetteva di posizionare gli accessori richiesti dai diversi modi di funzionamento (fig. 2.2):

[1] `http://nuclearheritage.com/projects/commercial-products`
[2] `http://www.ing.unipi.it/~a009435/issw/extra/therac.pdf`

- *specchio*: per il puntamento con un raggio di luce, **a fascio spento**.
- *elettromagneti e camera di ionizzazione*: per diffondere e misurare il fascio di **elettroni, ad energia bassa o alta ed a bassa intensità di corrente**.
- *bersaglio, diffusore e camera di ionizzazione*: per generare, diffondere e misurare il fascio di raggi **X, ad alta energia ed alta intensità di corrente**.

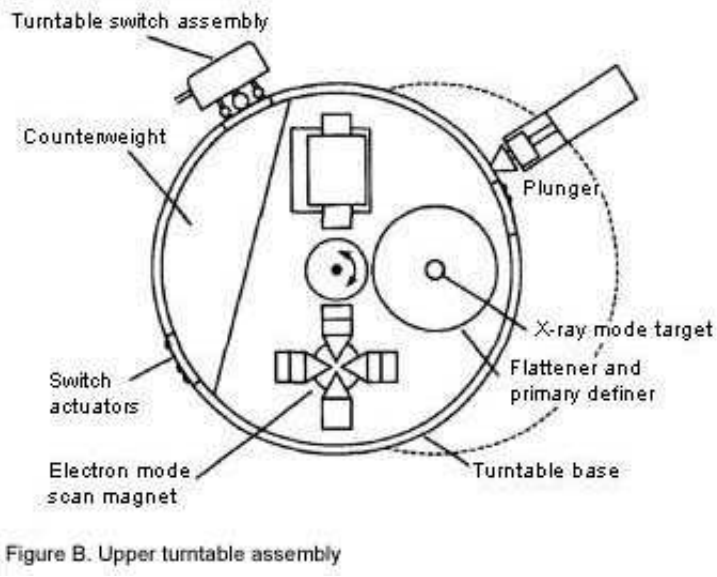


Figura 2.2: Il vassoio portaaccessori [25].

Il sovradosaggio avveniva quando il fascio di elettroni per raggi X (25 MeV ed alta intensità di corrente) era attivo senza che gli appositi accessori fossero posizionati.

Il Therac-25 era l'ultimo di una serie di prodotti per la radioterapia:

- il Therac-6, per trattamenti solo a raggi X, aveva blocchi di sicurezza elettromeccanici, a comando manuale (con interruttori, potenziometri e simili) e software (attraverso un terminale DEC VT100);
- il Therac-20, per trattamenti a raggi X ed elettroni, aveva blocchi di sicurezza elettromeccanici, a comando manuale e software.

Il Therac-25, per raggi X ed elettroni, aveva blocchi di sicurezza software e comando prevalentemente software; *i sensori letti dal processo software non avevano dei sistemi che ne controllassero il corretto funzionamento.*

L'operatore del Therac-25 usava un'interfaccia alfanumerica come quella mostrata in figura 2.3. Poteva inserire dei valori di default con un ritorno carrello e segnalare il completamento dell'inserimento o modifica dei dati portando il cursore sulla linea `COMMAND`. Il monitor mostra i parametri di funzionamento in due colonne:

- ACTUAL: valori rilevati dai sensori;

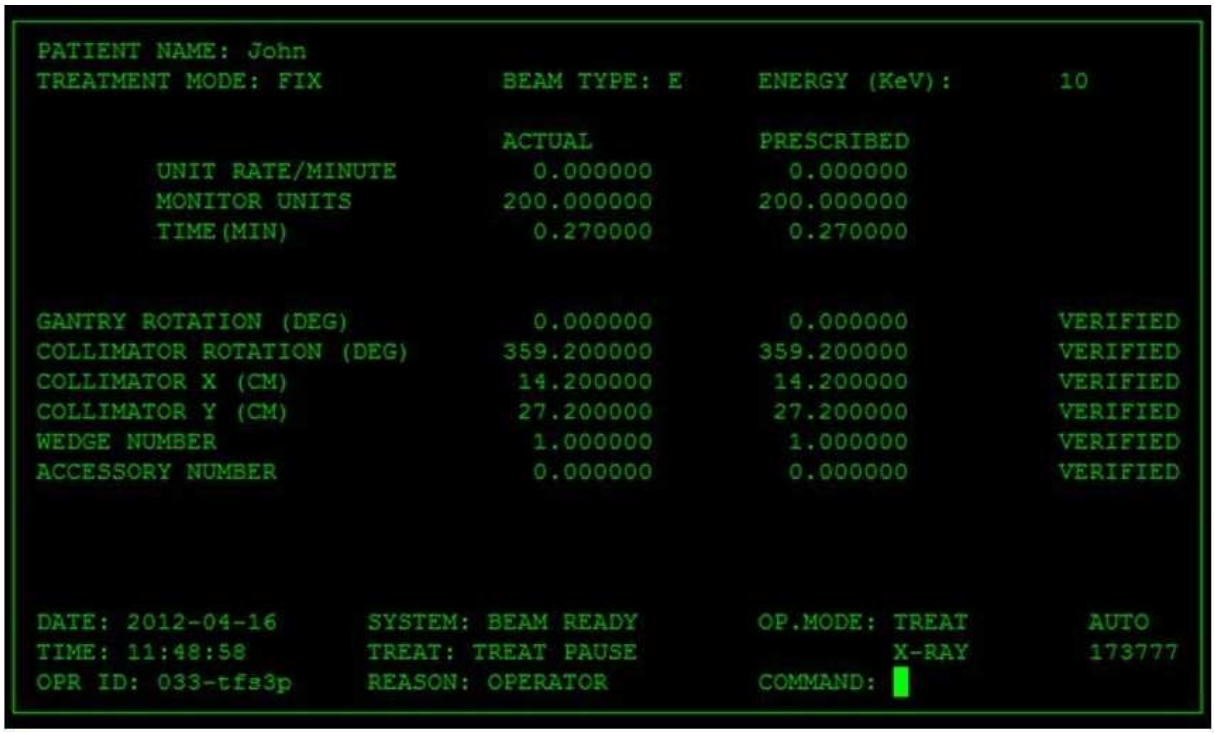


Figura 2.3: Interfaccia utente del Therac-25.

- PRESCRIBED: valori impostati dall'operatore.

Se i valori non corrispondono, sono possibili due condizioni di arresto:

- SUSPEND: richiede un reset completo (**reimpostando i parametri**) per riprendere il funzionamento;
- PAUSE: basta il comando **P** per riprendere, dopo aver corretto qualche parametro e **lasciando gli altri invariati**.

La gestione degli errori era del tutto insoddisfacente:

- molti messaggi consistevano solo in un numero, p.es., "`MALFUNCTION 54`";
- nessuna spiegazione nella documentazione;
- messaggi di significato ambiguo (e comunque non documentato):
  - `MALFUNCTION 54` → delivered dose **either too high or too low**;
- nessuna indicazione di livello di gravità;
- condizioni irrilevanti segnalate **frequentemente** come malfunzionamenti ("*al lupo, al lupo!*").

2.1.1 Gli incidenti dell'East Texas Cancer Center

Nel marzo del 1986 si verificò un primo incidente, che si può riassumere come segue:

- L'operatrice imposta velocemente i dati e porta il cursore su `COMMAND`;
- si accorge di aver scritto 'X' (raggi X) invece di 'E' (elettroni);
- corregge l'errore e usa il ritorno carrello per confermare gli altri dati;
- il display mostra `VERIFIED` e `BEAM READY`;
- l'operatrice accende il fascio (tasto **B**);
- la macchina va in `PAUSE` col messaggio `MALFUNCTION 54` e mostra una dose somministrata di 6 unità invece delle 202 richieste;
- l'operatrice preme **P**;
- `PAUSE`, `MALFUNCTION 54`, 6 unità;
- il paziente si alza dal tavolo e cerca di uscire dalla stanza.

Il paziente morí cinque mesi dopo.

Un altro incidente, nell'aprile seguente, avvenne in modo simile al precedente, ma l'operatrice non cercò di ripetere il trattamento dopo il primo arresto della macchina. Il paziente comunque morí tre settimane dopo.

Il problema software (a grandi linee)

Nel software di controllo ci sono diversi processi concorrenti, fra cui uno per gestire l'interfaccia e impostare i parametri ed uno per attivare i magneti di deflessione del fascio. Questi processi comunicano per mezzo di *variabili condivise*, ma *senza meccanismi di sincronizzazione* (semafori etc.). Quando la sequenza *impostazione-correzione-accensione* viene completata prima che finisca l'attivazione dei magneti, il sistema mantiene i valori non corretti, risultando nell'emissione di un fascio alla massima energia ed intensità con la macchina predisposta per il trattamento a elettroni.

Un problema hardware (a grandi linee)

Se la macchina è predisposta erroneamente per il trattamento a elettroni, la camera di ionizzazione, esposta ad un fascio per raggi X senza l'attenuazione del bersaglio e del diffusore, si satura e dà valori inattendibili (6 dosi nel caso considerato) e, come già accennato, il sistema non ha meccanismi per controllare che i sensori funzionino correttamente.

2.1.2 L'incidente dello Yakima Valley Memorial Hospital

Questo incidente avvenne nel gennaio 1987 ed è stato ricostruito in questo modo:

- L'operatore esegue due trattamenti a raggi X per un totale di 7 rad;
- imposta i dati per un trattamento a raggi X da 79 rad;
- usa lo specchio di puntamento e dà il comando **set** per posizionare correttamente il vassoio;
- il monitor mostra `BEAM READY` e l'operatore preme **B**;
- la macchina va in pausa e non mostra alcuna dose somministrata;

- l'operatore preme **P** e la macchina torna in pausa, mostrando una dose di 7 rad;
- il paziente si lamenta.

Il paziente morí in aprile.

Il problema software (a grandi linee)

Anche in questo caso ci sono due processi concorrenti che comunicano attraverso variabili condivise, in questo caso usate come variabili logiche per rappresentare diverse condizioni di funzionamento: una variabile chiamata `Class3` uguale a zero significa che i parametri sono corretti per il tipo di trattamento, ed una variabile chiamata `F$mal` uguale a zero significa che la macchina è pronta. Il processo `SetUpTest` incrementa `Class3` finché i parametri non sono corretti, poi legge `F$mal` e passa alla fase successiva se `F$mal` è uguale a zero. In modo concorrente, il processo `Lmtchk` legge `Class3` e se questa è uguale a zero *non* controlla la posizione del vassoio. Però `Class3` va a zero per *overflow* ogni 256 volte che viene incrementata, quindi può accadere che il fascio venga abilitato anche se il vassoio non è posizionato correttamente.

2.1.3 Le mancanze nel processo di sviluppo

Il caso del Therac-25 mette in evidenza un complesso di problemi nel processo di sviluppo che cercheremo di individuare.

Errori di programmazione

Le cause piú immediate dei malfunzionamenti stanno negli errori di programmazione, cioè difetti che si possono individuare nel codice sorgente. In particolare, sono stati individuati con certezza due errori: l'implementazione scorretta delle variabili logiche con il conseguente overflow, e la mancanza di meccanismi di protezione per le variabili condivise. Quest'ultimo problema verrà trattato approfonditamente in altri corsi, ma qualche accenno si trova piú oltre (sez. 6.3.3) in questa dispensa. L'altro problema sta nell'uso di una variabile di tipo intero per rappresentare condizioni logiche, con la convenzione di rappresentare il valore *vero* con lo zero ed il valore *falso* con qualsiasi numero diverso da zero. Questa convenzione non è sbagliata, ma è sbagliato assegnare il valore *falso* incrementando la variabile, perché questo può portare all'overflow, come si è visto.

Problemi di progetto e di sistema

Gli errori di programmazione visti nei paragrafi precedenti, ed altri errori probabilmente presenti ma non individuati, sono riconducibili a scelte di progetto di per sé accettabili, ma poco sicure.

Il sistema operativo in tempo reale era stato sviluppato *ad hoc*, mentre si sarebbe potuto usare il sistema DEC RT-11 fornito dal produttore del calcolatore adottato, il minicomputer PDP-11 della Digital Equipment Corporation. In generale, è consigliabile sfruttare prodotti software già disponibili, specialmente in applicazioni con elevati requisiti di sicurezza. Inoltre, il sistema era stato sviluppato interamente in assembler, mentre, col sistema operativo Unix, era stata dimostrata la possibilità di realizzare la maggior parte di un sistema operativo, riducendo al minimo indispensabile i moduli in assembler, in un linguaggio di piú alto livello come il C, che permette una maggior facilità di programmazione ed una migliore verificabilità.

Un altro problema, in parte collegato all'uso di un linguaggio a basso livello, era la mancanza di *programmazione difensiva*, cioè di meccanismi e accorgimenti (come, per esempio, l'uso di eccezioni ed asserzioni) per irrobustire il sistema rispetto a errori di programmazione e situazioni impreviste.

Un problema legato alla specifica del sistema piú che alla sua implementazione era la scelta di dotarlo di un'interfaccia utente piú amichevole che sicura: come già accennato esistevano diversi modi di accelerare le operazioni, per esempio confermando con un ritorno carrello i dati inseriti, ma sistemi di interazione piú rigidi avrebbero potuto evitare certi errori umani. Un altro grave difetto era la gestione degli errori, come descritta in precedenza.

Errori di processo

Le indagini sugli incidenti del Therac-25 misero in evidenza le mancanze nel processo di sviluppo del software, a cominciare dalla mancanza di documenti di specifica e progetto, oltre che di un piano di test e rapporti sui risultati dei test. Il controllo di qualità era stato insufficiente, e cosí le attività di collaudo. Fra l'altro, l'eccessiva complessità del software rendeva impossibile un collaudo sufficientemente esteso.

Un altro problema fu l'inadeguata analisi dei rischi, in cui era stata considerata trascurabile la probabilità di malfunzionamenti del software.

Infine, considerando l'attività di assistenza agli utenti dopo la consegna del sistema, fu osservata una tendenza a gestire con leggerezza le segnalazioni di problemi da parte degli utenti.

Errori di metodologia

Infine, possiamo considerare gli errori di metodologia, ovvero certi modi di affrontare lo sviluppo dei sistemi software, e anche dei sistemi integrati hardware-software (sistemi *embedded*), che sono all'origine degli errori di processo e di progetto.

L'inadeguatezza dell'analisi dei rischi, nel caso considerato, è legata alla sottovalutazione dei rischi dovuti al software. Nel periodo attuale la consapevolezza di tali rischi è cresciuta, ma è naturale che gli ingegneri elettronici o meccanici, nell'analisi di un sistema integrato, si concentrino sui rischi derivanti da guasti hardware. Tocca quindi all'ingegnere del software la responsabilità di valutare correttamente i rischi del software. Un altro

limite nell'analisi dei rischi nel caso del Therac-25 era quella che si potrebbe chiamare *valutazione pseudoquantitativa* dei rischi: le probabilità di eventi avversi venivano espresse numericamente, ma non venivano date giustificazioni sperimentali o teoriche per i valori indicati.

Inoltre, nel valutare la pericolosità del sistema, si confondeva l'*affidabilità* del sistema, cioè la probabilità di funzionare senza malfunzionamenti in un determinato periodo, con la *sicurezza*, cioè la probabilità di non avere mai dei malfunzionamenti capaci di causare danni gravi. Il Therac-25 era affidabile, avendo funzionato per lunghi periodi e in molte istallazioni senza problemi, ma non era sicuro, avendo avuto dei malfunzionamenti con esiti fatali.

Una confusione analoga era quella fra *uso* e *collaudo*. I produttori del sistema ritenevano che fosse stato collaudato a sufficienza perché era stato in uso per molto tempo. Ma l'uso di un sistema è il suo esercizio nel suo ambiente di applicazione, allo scopo di fornire i servizi richiesti, mentre il collaudo è l'esercizio del sistema in ambiente controllato, allo scopo di individuare malfunzionamenti.

Infine, un errore metodologico fu il riuso acritico del software sviluppato per sistemi precedenti. Il software del Therac-25 era in parte ripreso da quello del Therac-6 e Therac-20, ma questi sistemi avevano blocchi di sicurezza hardware che li proteggevano dai malfunzionamenti del software.

2.2 L'incidente Ariane 5 Flight 501

Un altro caso di malfunzionamento del software, che almeno non fece vittime ma ebbe costi molto alti, è quello del razzo del programma Ariane 5 che si distrusse 37 secondi dopo il lancio nel giugno del 1996 [7][3], [9][4]. Nonostante che questo incidente sia avvenuto circa dieci anni dopo quelli del Therac-25, le loro cause hanno molto in comune.

Il razzo disponeva di due *sistemi di riferimento inerziali* (SRI) per calcolare posizione, velocità ed accelerazione istantanee e passarle al calcolatore principale. I due SRI sono identici ed eseguono contemporaneamente lo stesso software, in modo che, in caso di guasto dell'SRI principale, quello di riserva possa subentrare immediatamente. I due SRI hanno una fase di inizializzazione che inizia alcuni secondi prima del lancio, dopo di che entrano nella fase di navigazione.

La distruzione del razzo fu causata da questa sequenza di eventi e condizioni:

- la conversione di un valore in virgola mobile su 64 bit in un valore intero su 16 bit sollevò un'eccezione di errore numerico nei due SRI;
- l'eccezione causò l'arresto degli SRI, che trasmisero un codice di errore al calcolatore;
- il codice di errore venne interpretato dal calcolatore principale come un dato valido;
- il calcolatore principale comandò un brusco cambiamento di rotta che portò il razzo alla distruzione.

[3] `http://www.ing.unipi.it/~a009435/issw/extra/esa-x-1819eng.pdf`
[4] `http://www.ing.unipi.it/~a009435/issw/extra/ariane5-benari.pdf`

L'errore numerico dipendeva dal fatto che il software dell'SRI era lo stesso usato nel precedente programma Ariane 4, in cui la conversione da valori in virgola mobile a valori interi avveniva correttamente. Ma nel programma Ariane 5, a causa delle diverse traiettorie previste, la variabile da convertire assumeva valori molto piú grandi, causando un errore di troncamento. A questo si aggiunge il fatto che l'operazione di conversione non era protetta dal meccanismo di gestione delle eccezioni.

Il software dell'Ariane 5 è scritto in assembler e in Ada. Quest'ultimo permette di scrivere del codice per gestire condizioni di errore rilevate da istruzioni di controllo inserite dal compilatore, ma è possibile anche eliminare tali istruzioni, se si pensa che una certa condizione di errore non si possa verificare. Il seguente frammento di codice[5] contiene le istruzioni relative all'operazione che causò l'arresto degli SRI:

```
declare
  horizontal_veloc_sensor: float;                      -- 1
  horizontal_veloc_bias: integer;                      -- 2
  ...
begin
  declare
    pragma suppress(numeric_error,
                    horizontal_veloc_bias);            -- 3
  begin
    sensor_get(horizontal_veloc_sensor);               -- 4
    horizontal_veloc_bias :=
                    integer(horizontal_veloc_sensor); -- 5
    ...
  exception
    when numeric_error => calculate_vertical_veloc(); -- 6
    when others => use_irs1();
  end;
```

Le istruzioni 1 e 2 sono le dichiarazioni delle variabili contenenti, rispettivamente, il valore della velocità orizzontale letta da un sensore (numero in virgola mobile su 64 bit) e lo stesso valore convertito in un intero su 16 bit. L'istruzione 3 sopprime i controlli sugli errori numerici nelle operazioni che coinvolgono la variabile `horizontal_veloc_bias`, l'istruzione 4 legge il sensore e la 5 esegue la conversione. L'istruzione 6 è il gestore di eccezioni che dovrebbe eseguire un codice alternativo in caso di errore nell'istruzione 5, ma questo gestore non verrà mai invocato perché sono state eliminate le istruzioni di controllo, per cui l'eccezione `numeric_error`, non essendo gestita, provoca l'arresto degli SRI.

L'evidente contraddizione fra la presenza di un gestore per l'eccezione `numeric_error` e di una direttiva che ne impedisce il rilevamento si spiega con la storia del processo di sviluppo: nel software del programma Ariane 4 tutte le variabili dell'SRI erano protette

[5] `http://www.sarahandrobin.com/ingo/swr/ariane5.html`

da gestori di eccezioni, ma quando il software è stato portato sui vettori Ariane 5 si è scelto di togliere la protezione ad alcune variabili, presumibilmente per motivi di efficienza computazionale e nella convinzione che la protezione non fosse necessaria.

Problemi di progetto e di sistema

A livello di progetto software, due problemi importanti sono la gestione delle eccezioni e l'interfaccia fra SRI e calcolatore principale. Come abbiamo visto, il controllo sull'eccezione `numeric_error` era stato soppresso perché, in base all'esperienza raccolta nelle missioni precedenti, si pensava che il problema non si presentasse. Quando invece l'eccezione si è presentata, il codice di errore presentato dall'SRI al calcolatore principale è stato interpretato come un valore di velocità, causando la distruzione del razzo.

Inoltre, il meccanismo di tolleranza ai guasti basato sulla doppia ridondanza è inefficace di fronte ai guasti software che, essendo sistematici e non casuali, si manifestano contemporaneamente nelle due unità SRI.

Errori di processo

Un errore fondamentale nel processo di sviluppo è la mancanza, nei documenti di specifica, della traiettoria prevista per il vettore. Da questa informazione sarebbe stato possibile scoprire che la conversione del valore della velocità orizzontale avrebbe causato un errore. Mancavano dalle specifiche anche i vincoli fisici del razzo, la cui conoscenza avrebbe potuto evitare il brusco cambiamento di rotta.

Si è anche appurato che le attività di collaudo (comprendenti il test del software e del sistema integrato, oltre alle simulazioni) erano insufficienti.

Errori di metodologia

Sebbene nel caso dell'Ariane 5 i problemi del software siano stati affrontati in modo molto piú sistematico che nel caso del Therac-25, tuttavia nel progetto della tolleranza ai guasti non si è tenuto conto, come visto piú sopra, della differenza fra guasti hardware, che sono casuali, e guasti software, che sono sistematici. In un sistema a doppia ridondanza, la tecnica della *diversità*, che consiste nell'eseguire software diversi sulle due unità ridondanti, potrebbe servire, ma nel caso dell'Ariane 5 i guasti non dipendono da errori di programmazione ma da specifiche incomplete.

All'incompletezza delle specifiche si aggiunge l'errore metodologico di riusare software preesistente in modo acritico, cioè senza indagare sulle differenze negli obiettivi (nuove traiettorie) e strumenti (nuovo vettore) rispetto alle missioni precedenti.

2.3 Sicurezza e sviluppo del software

Gli incidenti riportati in questo capitolo risalgono agli anni '80 e '90 del secolo scorso. Nel frattempo i campi di applicazione dei sistemi *safety-critical*, cioè capaci di provocare la morte di esseri umani in caso di guasto, sono diventati enormemente piú ampi e pervasivi. Pensiamo alla diffusione di dispositivi medici individuali, alcuni automatici come i pacemaker, altri controllabili dal paziente, come le pompe di infusione per l'insulina o gli analgesici. Pensiamo inoltre ai sistemi di assistenza alla guida di mezzi di trasporto collettivi e individuali, già evoluti al punto di poter sostituire il conducente umano. Pensiamo infine ai sistemi industriali per il controllo di impianti e macchine operatrici.

Nel tempo trascorso dagli incidenti del Therac-25 ad oggi, è cresciuta la consapevolezza di quanto sia importante la sicurezza dei sistemi software, ma purtroppo gli incidenti causati dal software continuano ad accadere, piú o meno per gli stessi motivi degli incidenti discussi in questo capitolo [26][6].

È indispensabile che ogni ingegnere del software si senta personalmente responsabile per la sicurezza del software che produce, e si impadronisca dei necessari strumenti teorici e pratici. Purtroppo tali strumenti non possono essere trattati in questo corso introduttivo. Qui possiamo solo accennare alle due strategie complementari da seguire:

prevenzione dei guasti (fault avoidance): la *qualità* del software è la prima linea di difesa contro i malfunzionamenti, poiché elimina molti difetti alla radice. Un'elevata qualità del software si ottiene attraverso processi di sviluppo rigorosi e l'uso metodico di collaudi, verifica e convalida. *Ça va sans dire*, questo presuppone esperienza e competenza da parte degli sviluppatori.

tolleranza ai guasti (fault tolerance): accorgimenti e tecniche per proteggere i sistemi, durante l'uso, da difetti di progetto o di programmazione sfuggiti alla convalida e verifica, o da situazioni impreviste. Queste tecniche vanno dalle elementari regole di programmazione difensiva all'adozione di standard di progetto orientati alla dependability [33][7],[43][8], all'uso di meccanismi di tolleranza ai guasti, come la *ridondanza* doppia o tripla, la *diversità*, la *correzione* degli errori, ed altre tecniche che vanno al di là dei limiti di questo corso.

[6] `http://www.ing.unipi.it/~a009435/issw/extra/therac30yrs_later.pdf`
[7] `http://www.ing.unipi.it/~a009435/issw/extra/misra-c-2004.pdf`
[8] `http://www.ing.unipi.it/~a009435/issw/extra/c-coding-standard.pdf`

Capitolo 3

Ciclo di vita e modelli di processo

Ogni prodotto industriale ha un *ciclo di vita* (fig. 3.1) che, a grandi linee, inizia quando si manifesta la necessità o l'utilità di un nuovo prodotto e prosegue con l'identificazione dei suoi requisiti, il progetto, la produzione, la verifica, e la consegna al committente. Dopo la consegna, il prodotto viene usato ed è quindi oggetto di manutenzione e assistenza tecnica, e infine termina il ciclo di vita col suo ritiro. A queste *attività* se ne aggiungono altre, che spesso vi si sovrappongono, come la pianificazione e la gestione del processo di sviluppo, e la documentazione. Ciascuna attività ne comprende altre, ovviamente in modo diverso per ciascun tipo di prodotto e per ciascuna organizzazione produttrice.

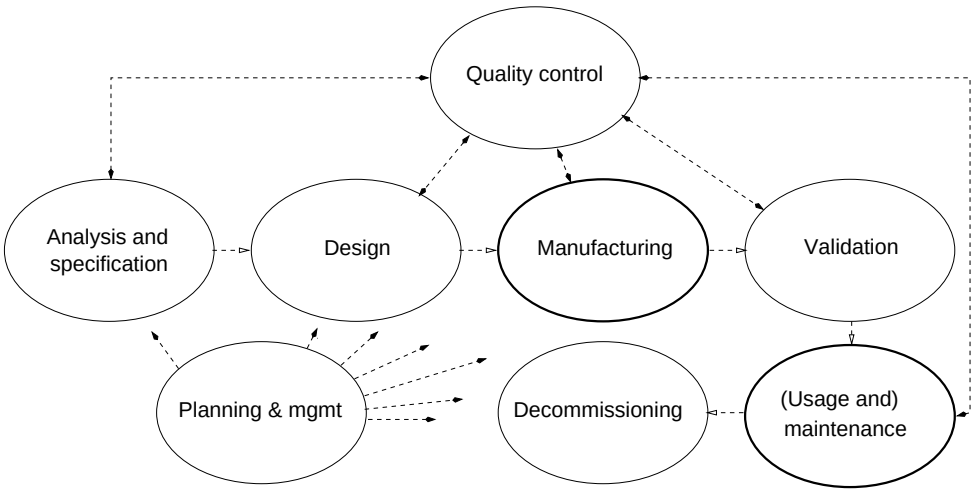


Figura 3.1: Il ciclo di vita di un prodotto industriale.

Un *processo di sviluppo* è un particolare modo di organizzare le attività costituenti il ciclo di vita, cioè di assegnare risorse alle varie attività e fissarne le scadenze. Una *fase* è un intervallo di tempo in cui si svolgono certe attività, e ciascuna attività può essere ripartita fra piú fasi. I diversi processi possono essere classificati secondo alcuni *modelli di processo*; un modello di processo è quindi una descrizione generica di una famiglia di processi simili, che realizzano il modello in modi diversi.

Esistono numerosi standard che definiscono processi di sviluppo o loro modelli, alcuni di applicabilità generale (per esempio, ISO/IEC 12207:2008 [2], MIL-STD-498 [32]), altri orientati a particolari settori di applicazione (per esempio, ESA PSS-05 [3] ed ECSS-E-40B [1] per l'industria spaziale, NS-G-1.1 [34] per l'industria nucleare, CEI EN 50128:2002-04 [5] per le ferrovie).

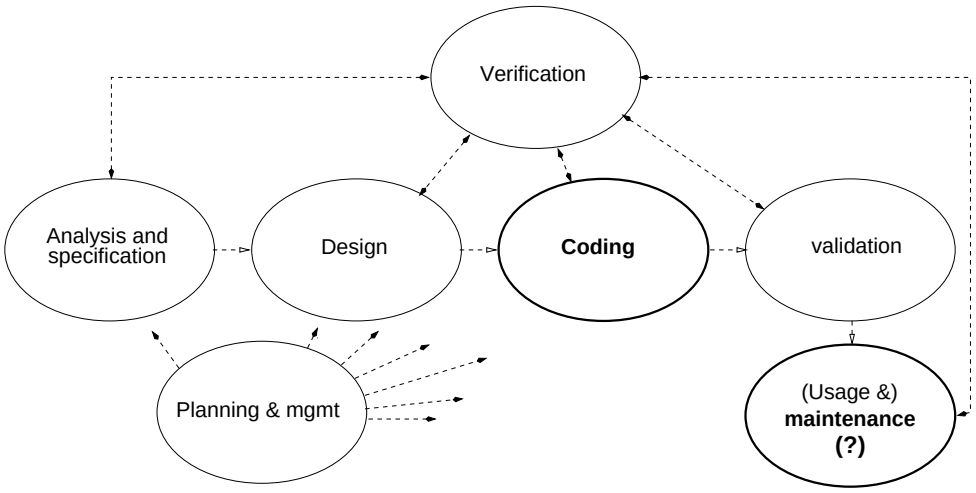


Figura 3.2: Il ciclo di vita del software.

Il ciclo di vita del software, di cui la fig. 3.2 mostra le cinque attività relative allo sviluppo del software in senso stretto (tralasciando le attività di consegna e di ritiro), segue lo schema generale appena esposto, ma con alcune importanti differenze, particolarmente nella fase di produzione. Nel software la riproduzione fisica del prodotto ha un peso economico ed organizzativo molto inferiore a quello che si trova nei prodotti tradizionali, per cui nel ciclo di vita del software il segmento corrispondente alla produzione è costituito dall'attività di programmazione, che, al pari delle fasi precedenti di analisi e di progetto, è un'attività di carattere intellettuale piuttosto che materiale. Un'altra importante differenza sta nella fase di manutenzione, che nel software ha un significato completamente diverso da quello tradizionale, come vedremo piú oltre.

Il ciclo di vita del software verrà studiato prendendo come esempio un particolare modello di processo, il modello a cascata, in cui ciascuna attività del ciclo di vita corrisponde ad una fase del processo (fig. 3.3). Successivamente si studieranno altri modelli di processo, in cui l'associazione fra attività e fasi del processo avviene in altri modi.

3.1 Il modello a cascata

Il *modello a cascata* (*waterfall*) prevede una successione di fasi consecutive. Ciascuna fase produce dei *semilavorati* (*deliverable*), cioè documenti relativi al processo, oppure documentazione del prodotto e codice sorgente o compilato, che vengono ulteriormente elaborati dalle fasi successive.

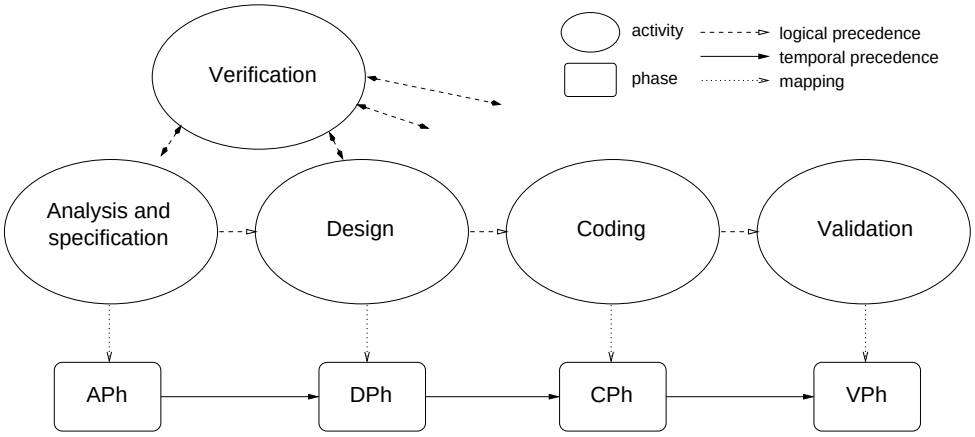


Figura 3.3: Un processo a cascata.

Il modello presuppone che ciascuna fase sia conclusa prima dell'inizio della fase successiva e che il flusso dei semilavorati sia, pertanto, rigidamente unidirezionale (come in una catena di montaggio): i risultati di una fase sono il punto di partenza della fase successiva, mentre non possono influenzare una fase precedente. Questo implica che in ogni fase si operi sul prodotto nella sua interezza: la fase di analisi produce le specifiche di tutto il sistema, quella di progetto produce il progetto di tutto il sistema. Quindi questo modello è adatto a progetti in cui i requisiti iniziali sono chiari fin dall'inizio e lo sviluppo del prodotto è prevedibile. Infatti, se in una certa fase si verificassero degli imprevisti, come la scoperta di errori od omissioni nel progetto o nelle specifiche, oppure l'introduzione di nuovi requisiti, allora si renderebbero necessarie la ripetizione delle fasi precedenti e la rielaborazione dei semilavorati prodotti fino a quel punto. Ma questa rielaborazione può essere molto costosa se la pianificazione del processo non la prevede fin dall'inizio: per esempio, il gruppo responsabile della fase di progetto potrebbe essere stato assegnato ad un altro incarico, o addirittura sciolto, all'inizio della fase di codifica. Inoltre, le modifiche sono rese costose dalle grandi dimensioni dei semilavorati.

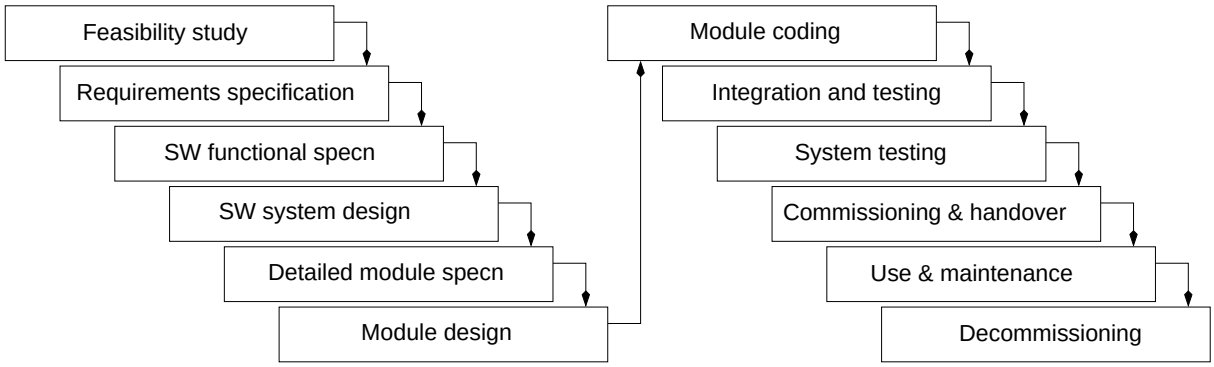


Figura 3.4: Il modello a cascata secondo [4].

Il numero, il contenuto e la denominazione delle fasi può variare da un'organizzazione all'altra, e da un progetto all'altro: la fig. 3.4, per esempio, rappresenta il processo di

sviluppo di software per le centrali nucleari proposto dalla International Atomic Energy Agency [4]. Nel séguito ci riferiremo ad un processo articolato nelle seguenti fasi:

- *studio di fattibilità*;
- *analisi e specifica dei requisiti*, suddivisa in
  - *analisi (definizione e specifica) dei requisiti dell'utente*, e
  - *specifica dei requisiti del software*;
- *progetto*, suddiviso in
  - *progetto architetturale*, e
  - *progetto in dettaglio*;
- *programmazione e test di unità*;
- *integrazione e test di sistema*;
- *manutenzione*.

Contemporaneamente a queste fasi, e nel corso di tutto il processo, si svolgono anche queste attività di supporto:

- *gestione*;
- *controllo di qualità*;
- *documentazione*.

Lo studio del modello a cascata è importante sia perché è il modello piú noto, sia perché l'analisi delle varie fasi permette di descrivere delle attività che fanno parte di tutti i modelli di processo, anche se raggruppate e pianificate in modi diversi.

3.1.1 Studio di fattibilità

Lo *studio di fattibilità* serve a stabilire se un dato prodotto può essere realizzato e se è conveniente realizzarlo, ad accertare quali sono le possibili strategie alternative per la sua realizzazione, a proporne un numero ristretto, a stimare la quantità di risorse necessarie, e quindi i costi relativi.

I metodi ed i criteri di questa fase dipendono sensibilmente dal rapporto fra committente e produttore: il prodotto di cui si vàluta la fattibilità può essere destinato alla stessa organizzazione di cui fa parte il produttore (p. es., il produttore vuole realizzare uno strumento CASE, *Computer Aided Software Engineering*, per uso interno), oppure ad un particolare committente esterno (il produttore realizza un database per un'azienda), oppure, genericamente, al mercato (il produttore realizza un database generico). Nei primi due casi è importante il dialogo fra produttore e committente (o fra sviluppatore e utente), che permette di chiarire i requisiti. Nel terzo caso il ruolo del committente viene assunto da quel settore dell'azienda che decide le caratteristiche dei nuovi prodotti.

In base al risultato dello studio di fattibilità, il committente decide se firmare o no il contratto per la fornitura del software. La rilevanza economica di questa fase può influenzare negativamente la qualità dello studio di fattibilità poiché il produttore di software,

per non perdere il contratto, può sottovalutare i costi e le difficoltà della proposta, ovvero sopravvalutare le proprie capacità e risorse. Questi errori di valutazione rischiano poi di causare ritardi e inadempienze contrattuali, con perdite economiche per il fornitore o per il committente.

A volte lo studio di fattibilità viene fornito come prodotto finito, indipendente dall'eventuale prosecuzione del progetto, che può essere abbandonato se il committente rinuncia, o essere affidato alla stessa organizzazione che ha fornito lo studio di fattibilità, oppure essere affidato ad un'altra organizzazione. Anche se il progetto viene abbandonato, lo studio di fattibilità è servito ad evitare il danno economico derivante dalla decisione di sviluppare un prodotto eccessivamente costoso o addirittura irrealizzabile. Inoltre, le conoscenze acquisite e rese accessibili nel corso dello studio di fattibilità contribuiscono ad arricchire il patrimonio di competenze del committente e del produttore.

Il semilavorato prodotto dallo studio di fattibilità è un documento che dovrebbe contenere queste informazioni:

- una descrizione del problema che deve essere risolto dall'applicazione, in termini di obiettivi e vincoli;
- un insieme di scenari possibili per la soluzione, sulla base di un'analisi dello stato dell'arte, cioè delle conoscenze e delle tecnologie disponibili;
- le modalità di sviluppo per le alternative proposte, insieme a una stima dei costi e dei tempi richiesti.

3.1.2 Analisi e specifica dei requisiti

Questa fase serve a capire e descrivere nel modo piú completo e preciso possibile *che cosa vuole* il committente dal prodotto software. La fig. 3.5 vuole dare un'immagine degli obiettivi dell'attività di analisi e specifica: un *modello di analisi* ed alcuni *semilavorati* che esprimono concretamente tale modello. Il modello in figura, estremamente semplificato, si riferisce al dominio di applicazione di un software per agenzie di viaggio ed è espresso nel linguaggio grafico UML, di cui si tratterà piú oltre (sez. 4.6). I semilavorati sono un certo numero di documenti che in vario modo descrivono il sistema da realizzare: la specifica dei requisiti del software definisce precisamente i servizi offerti dal software, il manuale utente descrive il sistema dal punto di vista dell'interazione con l'utente, e il piano di test specifica come il sistema deve rispondere a determinate azioni scelte per collaudarlo.

La fase di analisi e specifica può essere suddivisa nelle sottofasi di *analisi dei requisiti dell'utente* e *specifica dei requisiti del software*. La prima di queste sottofasi è rivolta alla comprensione del problema dell'utente e del contesto in cui dovrà operare il sistema software da sviluppare, richiede cioè una *analisi del dominio*, che porta all'acquisizione di conoscenze relative all'attività dell'utente stesso e all'ambiente in cui opera.

L'analisi dei requisiti dell'utente può essere ulteriormente suddivisa [44] in *definizione dei requisiti* e *specifica dei requisiti*: la definizione dei requisiti descrive ad alto livello servizi e vincoli mentre la specifica è piú dettagliata. Anche se la specifica dei requisiti contiene tutte le informazioni già fornite dalla definizione dei requisiti, a cui ne aggiunge

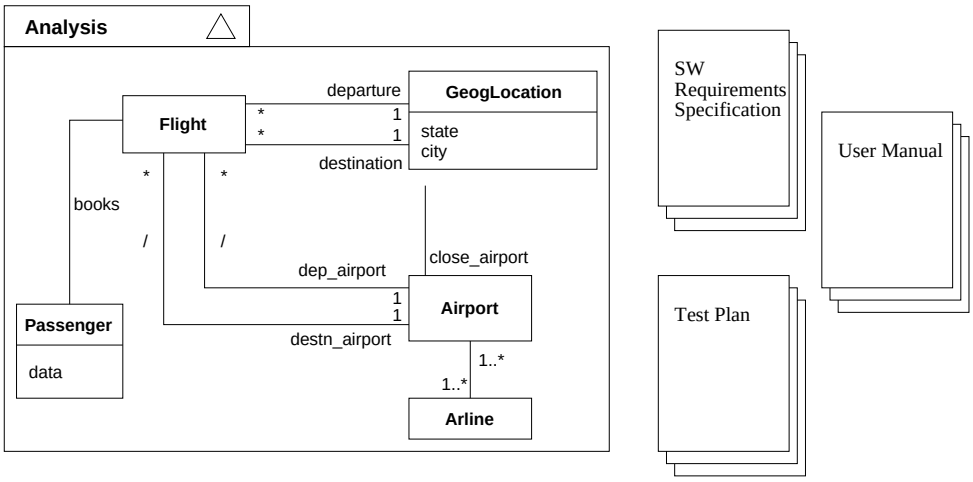


Figura 3.5: Un modello di analisi.

altre, sono necessari tutti e due i livelli di astrazione, in quanto la definizione dei requisiti, essendo meno dettagliata, permette una migliore comprensione generale del problema. Inoltre, la definizione e la specifica dei requisiti vengono usate da persone che hanno diversi ruoli e diverse competenze (committenti, amministratori, progettisti...), che richiedono diversi livelli di dettaglio.

Un livello di dettaglio ancora piú fine si ha nella specifica dei requisiti del software, che descrive le caratteristiche (*non* l'implementazione) del software che deve essere prodotto per soddisfare le esigenze dell'utente. Nella sottofase di specifica dei requisiti del software è importante evitare l'introduzione di scelte implementative, che in questa fase sono premature.

Come esempio di questi tre livelli di dettaglio, consideriamo il caso di uno strumento che permette di operare su file prodotti da altri strumenti, usando un'interfaccia grafica, come avviene, per esempio, col *desktop* di un PC (esempio adattato da [44]). Uno dei requisiti potrebbe essere espresso nei seguenti modi:

Analisi dei requisiti dell'utente

Definizione dei requisiti dell'utente

1 L'applicazione deve permettere la rappresentazione e l'elaborazione di file creati da altre applicazioni (detti *file esterni*).

Specifica dei requisiti dell'utente

1.1 L'applicazione deve permettere all'utente di definire i tipi dei file esterni.

1.2 Ad ogni tipo di file esterno corrisponde un programma esterno ed opzionalmente un'icona che viene usata per rappresentare il file. Se al tipo di un file non è associata alcuna icona, viene usata un'icona default non associata ad alcun tipo.

1.3 L'applicazione deve permettere all'utente di definire l'icona associata ai tipi di file esterni.

1.4 La selezione di un'icona rappresentante un file esterno causa l'elaborazione del file rappresentato, per mezzo del programma associato al tipo del file stesso.

Specifica dei requisiti del software

1.1.1 L'utente può definire i tipi dei file esterni sia per mezzo di menú che di finestre di dialogo. È opzionale la possibilità di definire i tipi dei file esterni per mezzo di file di configurazione modificabili dall'utente.

1.2.1 L'utente può associare un programma esterno ad un tipo di file esterno sia per mezzo di finestre di dialogo che di file di configurazione.

1.2.2 L'utente può associare un'icona ad un tipo di file esterno per mezzo di una finestra di selezione grafica (*chooser*).

1.3.1 L'applicazione deve comprendere una libreria di icone già pronte ed uno strumento grafico che permetta all'utente di crearne di nuove.

1.4.1 La selezione di un'icona rappresentante un file esterno può avvenire sia per mezzo del mouse che della tastiera.

Osserviamo lo schema di numerazione dei requisiti, che permette di collegare fra di loro requisiti a diverso livello di astrazione: per esempio, i requisiti del software 1.2.1 e 1.2.2 sono raffinamenti del requisito utente 1.2 (a livello di specifica dei requisiti) che a sua volta è un raffinamento del requisito utente 1 (a livello di definizione dei requisiti). Questo semplice espediente facilita la *tracciabilità* dei requisiti, cioè la possibilità di giustificare l'introduzione di ciascun requisito in termini di requisiti piú generali.

Requisiti funzionali e non funzionali

I requisiti possono essere *funzionali* o *non funzionali*. I requisiti funzionali descrivono cosa deve fare il prodotto, generalmente in termini di relazioni fra dati di ingresso e dati di uscita, mentre i requisiti non funzionali sono caratteristiche di qualità come, per esempio, l'affidabilità o l'usabilità, oppure vincoli di varia natura. Di questi requisiti si parlerà piú diffusamente in séguito.

Altri requisiti possono riguardare il processo di sviluppo anziché il prodotto. Per esempio, il committente può richiedere che vengano applicate determinate procedure di controllo di qualità o vengano seguiti determinati standard.

Documenti di specifica

Il prodotto della fase di analisi e specifica dei requisiti generalmente è costituito da questi documenti:

documento di specifica dei requisiti: il Documento di Specifica dei Requisiti (DSR) è il fondamento di tutto il lavoro successivo, e, se il prodotto è sviluppato per un committente esterno, ha pure un valore legale poiché viene incluso nel contratto.

manuale utente: descrive il comportamento del sistema dal punto di vista dell'utente ("se tu fai questo, succede quest'altro").

piano di test di sistema: definisce come verranno eseguiti i test finali per convalidare il prodotto rispetto ai requisiti. Anche questo documento può avere valore legale, se firmato dal committente, che cosí accetta l'esecuzione del piano di test come collaudo per l'accettazione del sistema.

Il DSR deve riportare almeno queste informazioni [18]:

- una descrizione del dominio dell'applicazione da sviluppare, comprendente l'individuazione delle parti in causa e delle entità costituenti il dominio (persone, oggetti materiali, organizzazioni, concetti astratti...) con le loro relazioni reciproche;
- gli scopi dell'applicazione;
- i requisiti funzionali;
- i requisiti non funzionali;
- i requisiti sulla gestione del processo di sviluppo.

I documenti di specifica devono essere scritti in un linguaggio il piú possibile preciso e non ambiguo. La questione dei linguaggi di specifica verrà trattata piú oltre (sez. 4.3) e qui osserviamo soltanto che conviene usare il piú possibile dei linguaggi *formali*, tali cioè da poter essere interpretati secondo regole rigorose, analogamente al linguaggio matematico.

Esempi di DSR si trovano alla voce *Laboratorio* sulla pagina web (`http://www2.ing.unipi.it/~a009435/issw/isw.html`) del corso.

3.1.3 Progetto

In questa fase si stabilisce *come* deve essere fatto il sistema definito dai documenti di specifica (DSR e manuale utente). Poiché, in generale, esistono diversi modi di realizzare un sistema che soddisfi un insieme di requisiti, l'attività del progettista consiste essenzialmente in una serie di *scelte* fra le soluzioni possibili, guidate da alcuni princípi e criteri che verranno illustrati nei capitoli successivi.

Il risultato del progetto è una *architettura software*, cioè una scomposizione del sistema in elementi strutturali, detti *moduli*, dei quali vengono specificate le funzionalità e le relazioni reciproche. La fase di progetto può essere suddivisa nelle sottofasi di *progetto architetturale* e di *progetto in dettaglio*. Nella prima fase viene definita la struttura generale del sistema, mentre nella seconda si definiscono i singoli moduli. La distinzione fra queste due sottofasi spesso non è netta, e nelle metodologie di progetto piú moderne tende a sfumare.

Il principale semilavorato prodotto da questa fase è il *Documento delle Specifiche di Progetto* (DSP).

Anche il DSP dovrebbe poter essere scritto in modo rigoroso ed univoco, possibilmente usando notazioni formali (*linguaggi di progetto*). In pratica ci si affida prevalentemente al linguaggio naturale integrato con notazioni grafiche.

In questa fase può essere prodotto anche il *Piano di Test di Integrazione*, che prescrive come collaudare l'interfacciamento fra i moduli nel corso della costruzione del sistema (Sez. 7.5.1).

3.1.4 Programmazione (codifica) e test di unità

In questa fase i singoli moduli definiti nella fase di progetto vengono implementati e collaudati singolarmente. Vengono scelte le strutture dati e gli algoritmi, che di solito non vengono specificati dal DSP.

In questo corso non si parlerà della programmazione, di cui si suppongono noti i princípi e le tecniche, ma si indicheranno le caratteristiche dei linguaggi orientati agli oggetti che permettono di applicare alcuni concetti relativi al progetto del software. Inoltre vogliamo accennare in questa sezione ad alcuni aspetti del lavoro di programmazione e ad alcuni strumenti relativi:

gestione delle versioni: Durante la programmazione vengono prodotte numerose versioni dei componenti software, ed è importante conservare e gestire tali versioni. Uno strumento molto diffuso è *Subversion* (*SVN*[1]), che permette di gestire un archivio (*repository*[2]) del codice sorgente a cui gli sviluppatori possono accedere in modo concorrente, anche da locazioni remote. Lo strumento di questo tipo attualmente piú diffuso è probabilmente il *git*[3].

configurazione e compilazione automatica: Esistono strumenti che permettono di automatizzare il processo di costruzione (*build*) (compilazione e collegamento) del software, e di configurare questo processo, cioè di adattarlo a diverse piattaforme software. Gli *ambienti di sviluppo integrati* (*integrated development environments*, IDE) svolgono queste funzioni attraverso un'interfaccia grafica che permette di creare e modificare i file sorgente e compiere tutte le operazioni di costruzione, ma spesso è utile ricorrere a strumenti *a linea di comando* che possono offrire allo sviluppatore un controllo piú fine sul processo di costruzione. Alcuni di questi strumenti sono i programmi *Make*, *Automake*, *Autoconf*, e *Libtool*, noti collettivamente come *Autotools*[4].

documentazione del codice: È possibile produrre la documentazione del codice usando strumenti come *Doxygen*[5], *Javadoc*[6] e simili. Questi strumenti estraggono dal codice sorgente i commenti inseriti dal programmatore e producono documenti in HTML o PDF.

notifica e archiviazione di malfunzionamenti: Gli strumenti per il *bug tracking*, per esempio *Bugzilla*[7], permettono di segnalare agli sviluppatori i guasti rilevati nell'uso del software, di archiviare tali notifiche, e di tenere utenti e sviluppatori al corrente sui progressi nell'attività di *debugging*.

test di unità: Il test di unità può essere parzialmente automatizzato grazie a strumenti come *DejaGNU*[8], *CppUnit*[9], *Mockpp*[10]. Gli ultimi due verranno trattati nel cap. 7.

[1] `http://subversion.tigris.org`
[2] Si pronuncia con l'accento tonico sulla seconda sillaba.
[3] `http://git-scm.com`
[4] `http://www.gnu.org/manual`
[5] `http://www.doxygen.org`
[6] `http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html`
[7] `http://www.mozilla.org/projects/bugzilla`
[8] `http://www.gnu.org/software/dejagnu`
[9] `http://cppunit.sourceforge.net`
[10] `http://mockpp.sourceforge.net`

Il prodotto della fase di programmazione e test di unità è costituito dal codice dei programmi con la relativa documentazione e dalla documentazione relativa ai test.

3.1.5 Integrazione e test di sistema

In questa fase viene messo assieme e collaudato il prodotto completo. Il costo di questa fase ovviamente è tanto maggiore quanto maggiori sono le dimensioni e la complessità dell'applicazione. Specialmente se l'applicazione viene sviluppata da gruppi di lavoro diversi, questa fase richiede un'accurata pianificazione. Il lavoro svolto in questa fase viene tanto piú facilitato quanto piú l'applicazione è *modulare*, concetto che verrà introdotto nel cap. 5.

Nel corso dell'integrazione vengono assemblati i vari sottosistemi a partire dai moduli componenti, effettuando parallelamente il *test di integrazione* (sez. 7.5.1), che verifica la corretta interazione fra i moduli. Dopo che il sistema è stato assemblato completamente, viene eseguito il *test di sistema* (sez. 7.5.2).

Il test di sistema può essere seguito, quando l'applicazione è indirizzata al mercato, dall'*alfa test*, in cui l'applicazione viene usata all'interno all'azienda produttrice, e dal *beta test*, in cui l'applicazione viene usata da pochi utenti esterni selezionati (*beta tester*).

3.1.6 Manutenzione

Il termine "manutenzione" applicato al software è improprio, poiché il software non soffre di usura o invecchiamento e non ha bisogno di rifornimenti. La cosiddetta manutenzione del software è in realtà la correzione di errori presenti nel prodotto consegnato al committente, oppure l'aggiornamento del codice allo scopo di fornire nuove versioni. Questa attività consiste nel modificare e ricostruire il software. Si dovrebbe quindi parlare di *riprogettazione* piuttosto che di manutenzione. La scoperta di errori dovrebbe portare ad un riesame critico del progetto e delle specifiche, ma spesso nella prassi comune questo non avviene, particolarmente quando si adotta il modello a cascata. La manutenzione avviene piuttosto attraverso un "rattoppo" (*patch*[11]) del codice sorgente. Questo fa sí che il codice non corrisponda piú al progetto, per cui aumenta la difficoltà di correggere ulteriori errori. Dopo una serie di operazioni di manutenzione, il codice sorgente può essere talmente degradato da perdere la sua struttura originaria e qualsiasi relazione con la documentazione. In certi casi diventa necessario ricostruire il codice con interventi di *reingegnerizzazione* (*reengineering* o *reverse engineering*). L'uso di strumenti per la gestione delle versioni rende molto piú facile e controllabile l'attività di manutenzione.

Per contenere i costi e gli effetti avversi della manutenzione è necessario tener conto fin dalla fase di progetto che sarà necessario modificare il software prodotto. Questo

[11] "*I can recall the days when a programmer would 'patch' a program stored on paper tape by using glue and paper.*" – D.L. Parnas, *Software aging*, 1994 [38]

principio è chiamato "progettare per il cambiamento" (*design for change* [37]). Anche qui osserviamo che la modularità del sistema ne facilita la modifica.

Si distinguono i seguenti tipi di manutenzione:

correttiva: individuare e correggere errori.

adattativa: cambiamenti di ambiente operativo (*porting*) in senso stretto, cioè relativo al cambiamento di piattaforma hardware e software, ma anche in senso piú ampio, relativo a cambiamenti di leggi o procedure, linguistici e culturali (per esempio, le date si scrivono in modi diversi in diversi paesi)[12].

perfettiva: aggiunte e miglioramenti.

3.1.7 Attività di supporto

Alcune attività vengono svolte contemporaneamente alle fasi già illustrate:

gestione: la gestione comprende la pianificazione del processo di sviluppo, la allocazione delle risorse (in particolare le risorse umane, con lo *staffing*), l'organizzazione dei flussi di informazione fra i gruppi di lavoro e al loro interno, ed altre attività di carattere organizzativo. Un aspetto particolare della gestione, orientato al prodotto piú che al processo, è la *gestione delle configurazioni*, volta a mantenere i componenti del prodotto e le loro versioni in uno stato aggiornato e consistente. Per *configurazione* di un prodotto si intende l'elenco dei suoi componenti, per ciascuno dei quali deve essere indicata la versione richiesta.

documentazione: la maggior parte dei deliverable è costituita da documentazione. Altra documentazione viene prodotta ed usata internamente al processo di sviluppo, come, per esempio, rapporti periodici sull'avanzamento dei lavori, linee guida per gli sviluppatori, minute delle riunioni, e simili. Gli strumenti per la gestione delle versioni del codice sorgente possono essere usate anche per mantenere la documentazione, ma esistono anche strumenti destinati specificamente alla documentazione. In particolare si possono usare strumenti orientati alla collaborazione ed alla condivisione delle informazioni, come le pagine *wiki*[13].

convalida e verifica: la correttezza funzionale e l'adeguatezza ai requisiti, sia del prodotto finale che dei semilavorati, devono essere accertati.

controllo di qualità: oltre ai controlli finali sul prodotto, occorre controllare ed assicurare la qualità del processo di sviluppo nel suo complesso, verificando che le varie attività rispettino gli standard e le procedure previste.

Convalida e verifica

La coppia di termini *convalida* (*validation*) e *verifica* (*verification*) viene usata in letteratura con due accezioni leggermente diverse. Per comprendere la differenza, bisogna chiarire i concetti di *requisiti* e *specifiche*: i requisiti sono ciò che il committente e l'utente

[12] `http://www.gnu.org/software/gettext/`

[13] `http://en.wikipedia.org/wiki/Wiki`

si aspettano dal prodotto software, e sono espressi in modo informale ed approssimativo, a volte restando impliciti e sottintesi; le specifiche sono l'espressione esplicita e rigorosa dei requisiti (fig. 3.6). Osserviamo però che il termine "requisiti" si usa anche per riferirsi ai singoli elementi del DSR.

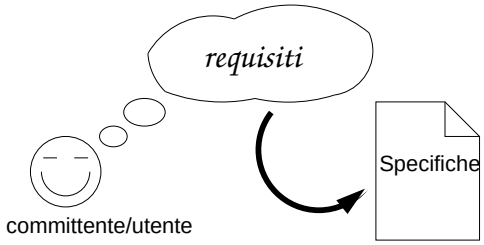


Figura 3.6: Requisiti e specifiche.

La stesura delle specifiche è quindi un lavoro di interpretazione dei requisiti, e non si può dare per scontato che le specifiche rispecchino fedelmente i requisiti, cioè le reali esigenze del committente o utente, come mostra una famosa vignetta riproposta con innumerevoli variazioni nella letteratura dell'ingegneria del software (fig. 3.7, da [19]).

In una delle due accezioni usate, la convalida consiste nel valutare la correttezza e la qualità del prodotto rispetto ai requisiti, mentre la verifica prende come termine di riferimento le specifiche. In questa accezione, sia la convalida che la verifica si possono applicare in piú fasi del processo; in particolare, ogni semilavorato del processo a cascata si considera come specifica per quello successivo e come implementazione di quello precedente (salvo, ovviamente, il primo e l'ultimo), per cui ogni semilavorato viene sottoposto a verifica. La convalida si applica al prodotto finito, ma può essere applicata anche ai documenti di specifica, a prototipi o a implementazioni parziali dell'applicazione.

Nell'altra accezione, si intende per convalida soltanto la valutazione del prodotto *finale* rispetto ai requisiti, mentre la verifica si applica ai semilavorati di ciascuna fase intermedia.

3.2 Il modello di processo a V

Il *modello di processo a V* è una variante del modello a cascata in cui si mettono in evidenza le fasi di collaudo e la loro relazione con le fasi di sviluppo precedenti. La fig. 3.8 mostra un esempio di processo a V, dallo standard IAEA TRS 384 [4].

In questo particolare processo si intende per "convalida" la valutazione del sistema finale. Inoltre questo processo descrive lo sviluppo del sistema integrato (*sistema*) costituito da hardware e software (*sistema di elaborazione*, *computer system*).

La rappresentazione a V mostra come la fase di test di integrazione del sistema di elaborazione sia guidata dalle specifiche del sistema di elaborazione, e come la fase di test di convalida di sistema sia guidata dalla specifica dei requisiti di sistema.

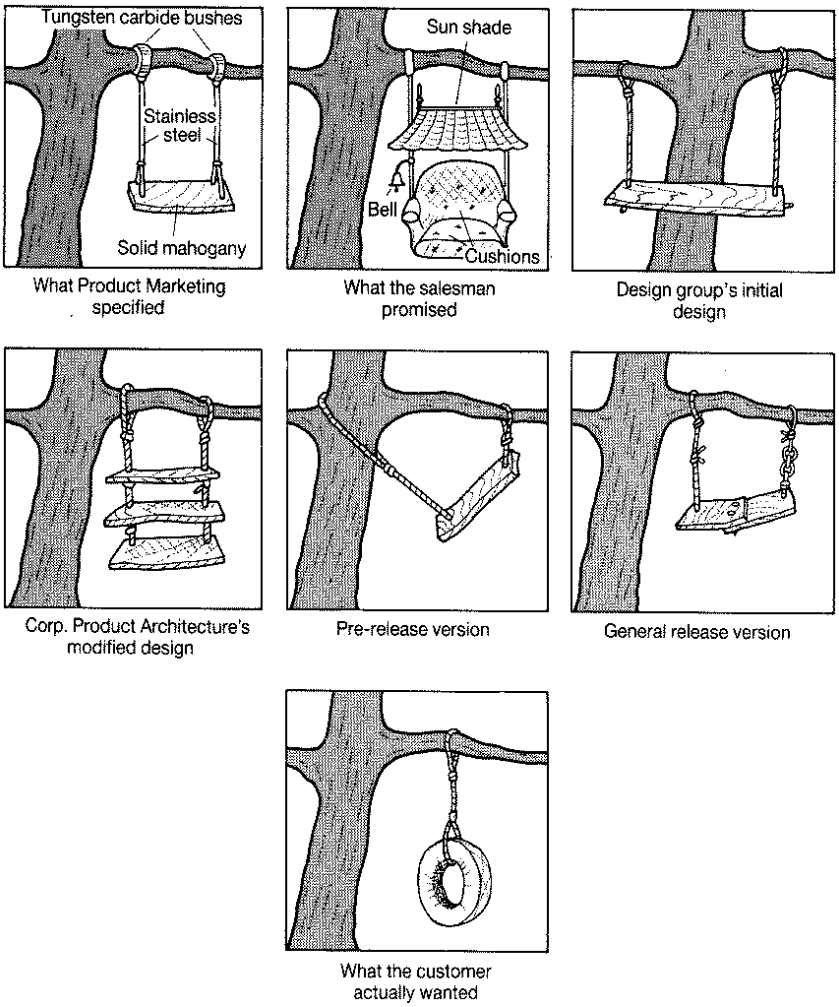


Figura 3.7: Il processo di sviluppo del software [19].

Osserviamo che, in base ai risultati delle attività di convalida, è possibile ripetere le fasi di specifica, per cui il modello a V può essere considerato come un modello intermedio fra il modello a cascata puro ed i modelli iterativi che vedremo piú oltre.

Un altro esempio [45] di processo a V è mostrato in fig. 3.9. Questo processo descrive specificamente lo sviluppo del sistema software.

3.3 Modelli evolutivi

Abbiamo osservato che nello sviluppo del software bisogna prevedere la necessità di cambiamenti. È quindi opportuno usare processi di sviluppo in cui la necessità di introdurre dei cambiamenti venga rilevata tempestivamente ed i cambiamenti stessi vengano introdotti facilmente.

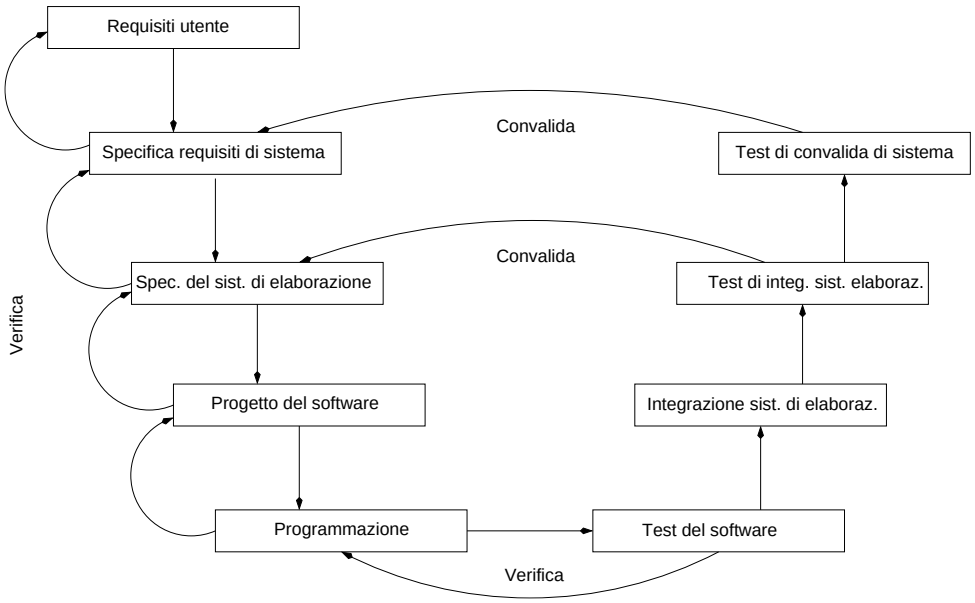


Figura 3.8: Un processo a V (da IAEA TRS 384, modificato).

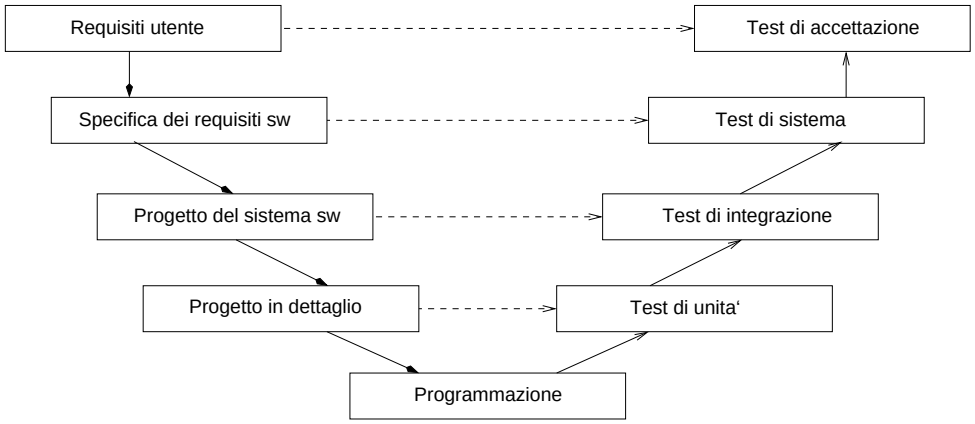


Figura 3.9: Un processo a V (da [45]).

Nei processi basati su modelli evolutivi, il software viene prodotto in modo incrementale, in passi successivi (fig. 3.10). Ogni passo produce, a seconda delle varie strategie possibili, una parte nuova oppure una versione via via piú raffinata e perfezionata del sistema complessivo. Il prodotto di ciascun passo viene valutato ed i risultati di questa valutazione determinano i passi successivi, finché non si arriva ad un sistema che risponda pienamente alle esigenze del committente, almeno finché non si sentirà il bisogno di una nuova versione.

3.3.1 Prototipazione

Un *prototipo* è una versione approssimata, parziale, ma funzionante, dell'applicazione che viene sviluppata. La prototipazione, cioè la costruzione e l'uso di prototipi, entra nei

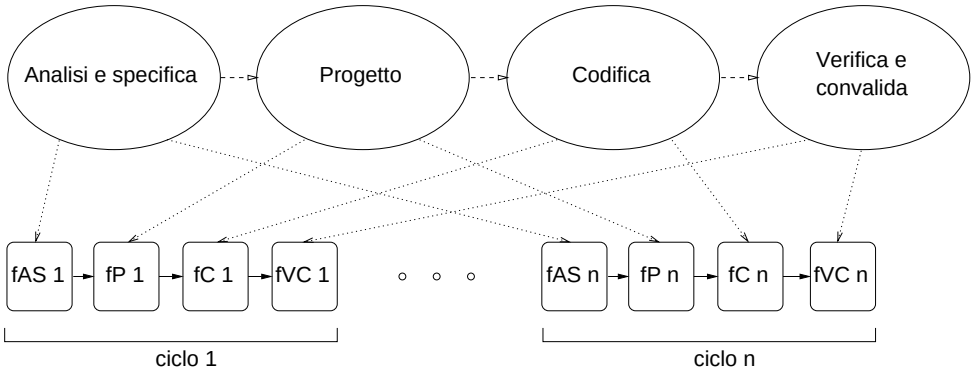


Figura 3.10: Un processo incrementale.

modelli evolutivi in diversi modi [28]:

- Un prototipo può essere costruito e valutato nel corso dello studio di fattibilità.
- Un prototipo *esplorativo* viene costruito e consegnato all'utente durante l'analisi dei requisiti, in modo che l'utente possa provarlo e chiarire i requisiti del sistema.
- Un prototipo *sperimentale* viene usato nella fase di progetto per studiare diverse implementazioni alternative.
- Un prototipo *evolutivo* è una parte funzionante ed autonoma del sistema finale, e può essere consegnato all'utente, che comincia ad usarlo nella propria attività (strategia "*early subset, early delivery*" o *incremental delivery* [17]).

Quando il prototipo viene usato nell'analisi dei requisiti, bisogna scegliere se buttar via il prototipo, una volta ottenuta l'approvazione dell'utente, per passare al progetto *ex novo* del sistema, oppure costruire il prodotto finale ampliando e perfezionando il prototipo. Nel primo caso si parla di prototipo *usa e getta* (*throw-away*), nel secondo si parla di prototipo *evolutivo* o *sistema pilota*.

Un prototipo usa e getta ha quindi una struttura interna del tutto diversa da quello che sarà il prodotto finale, e può essere realizzato senza vincoli di efficienza, per esempio usando linguaggi dichiarativi (come, p.es., il Prolog [11]) o linguaggi di scripting (p.es., Tcl/Tk o Python) che permettono tempi di sviluppo piú brevi a scapito delle prestazioni e di altre qualità desiderabili del prodotto.

Un prototipo evolutivo richiede che fin dall'inizio si facciano scelte che condizioneranno le fasi successive. Questo richiede naturalmente un maggiore sforzo di progetto. Un rischio comune nell'uso dei prototipi consiste nel "prendere sul serio" delle scelte che in fase di prototipazione erano accettabili e trascinarle fino alla realizzazione del sistema, anche se inadeguate al prodotto finale. Per esempio, durante la prototipazione si può scegliere un algoritmo inefficiente per una determinata funzione, in attesa di trovarne uno migliore; ma una volta che il prototipo è fatto e "funziona" (piú o meno bene), è facile cedere alla tentazione di prendere per buono quello che è stato fatto e lasciare nel prodotto un componente di cui fin dall'inizio era nota l'inadeguatezza.

Infine, osserviamo che una forma molto comune di prototipazione consiste nello sviluppo dell'interfaccia utente. A questo scopo sono disponibili ambienti, linguaggi e librerie che permettono di realizzare facilmente delle interfacce interattive.

3.3.2 Lo Unified Process

Lo Unified Process (UP) [8, 23] è un processo evolutivo per lo sviluppo di software orientato agli oggetti, concepito dagli ideatori del linguaggio UML. Questo processo si può schematizzare come segue:

- l'arco temporale del processo di sviluppo è suddiviso in quattro *fasi* successive;
- ogni fase ha un obiettivo e produce un insieme di semilavorati chiamato *milestone* ("pietra miliare");
- ogni fase è suddivisa in un numero variabile di *iterazioni*;
- nel corso di ciascuna iterazione possono essere svolte tutte le attività richieste (analisi, progetto...), anche se, a seconda della fase e degli obiettivi dell'iterazione, alcune attività possono essere predominanti ed altre possono mancare;
- ciascuna iterazione produce una versione provvisoria (*baseline*) del prodotto, insieme alla documentazione associata.

Attività (*workflow*)

Nello UP si riconoscono cinque attività, dette *workflow* (o *flussi di lavoro*): *raccolta dei requisiti* (*requirement capture*), *analisi* (*analysis*), *progetto* (*design*), *implementazione* (*implementation*), e *collaudo* (*test*). Le prime due attività corrispondono a quella che abbiamo chiamato complessivamente *analisi e specifica dei requisiti*: piú precisamente, la raccolta dei requisiti porta alla definizione di un *modello dei casi d'uso* che rappresenta le funzioni ed i servizi offerti all'utente, mentre l'attività di analisi produce un *modello di analisi* che rappresenta i concetti e le entità del dominio di applicazione rilevanti per il prodotto da sviluppare.

Fasi

Le quattro fasi dello UP sono:

inizio (*inception*): i suoi obiettivi corrispondono a quelli visti per lo studio di fattibilità, a cui si aggiunge una analisi dei rischi di varia natura (tecnica, economica, organizzativa...) in cui può incorrere il progetto. Anche il *milestone* di questa fase è simile all'insieme di documenti e altri artefatti (per esempio, dei prototipi) che si possono produrre in uno studio di fattibilità. Un documento caratteristico dello UP, prodotto in questa fase, è il *modello dei casi d'uso*, cioè una descrizione sintetica delle possibili interazioni degli utenti col sistema, espressa mediante la notazione UML.

elaborazione (*elaboration*): gli obiettivi di questa fase consistono nell'estendere e perfezionare le conoscenze acquisite nella fase precedente, e nel produrre una *baseline architetturale eseguibile*. Questa è una prima versione, eseguibile anche se parziale, del sistema, che non si deve considerare un prototipo, ma una specie di ossatura che serva da base per lo sviluppo successivo. Il milestone della fase comprende quindi il codice che costituisce la baseline, il suo modello costituito da vari diagrammi UML, e le versioni aggiornate dei documenti prodotti nella fase di inizio.

costruzione (*construction*): ha l'obiettivo di produrre il sistema finale, partendo dalla baseline architetturale e completando le attività di raccolta dei requisiti, analisi e progetto portate avanti nelle fasi precedenti. Il milestone comprende, fra l'altro, il sistema stesso, la sua documentazione in UML, una *test suite* e i manuali utente. La fase si conclude con un periodo di beta test.

transizione (*transition*): gli obiettivi di questa fase consistono nella correzione degli errori trovati in fase di beta test e quindi nella consegna e messa in opera (*deployment*) del sistema. Il milestone consiste nella versione definitiva del sistema e dei manuali utente, e nel piano di assistenza tecnica.

Distribuzione delle attività nelle fasi

Come risulta dai paragrafi precedenti, in ciascuna fase si possono svolgere attività diverse. Nella fase di inizio sono preponderanti le attività di raccolta e analisi dei requisiti, ma c'è una componente di progettazione per definire un'architettura iniziale ad alto livello, non eseguibile. Può essere richiesta anche l'attività di implementazione, se si realizzano dei prototipi. Nella fase di elaborazione le cinque attività tendono ad avere pesi simili nello sforzo complessivo, con la tendenza per le attività di analisi a decrescere nelle iterazioni finali in termini di lavoro impegnato, mentre le attività di progetto e implementazione crescono corrispondentemente. Nella fase di costruzione sono preponderanti le attività di progetto e implementazione, ma non sono ancora terminate quelle di analisi, essendo previsto, come in tutti i processi evolutivi, che i requisiti possano cambiare in qualunque momento. Naturalmente, un accurato lavoro di analisi dei requisiti nelle fasi iniziali renderà poco probabile l'eventualità di grandi cambiamenti nelle fasi finali, per cui ci si aspetta che in queste ultime i cambiamenti siano limitati ed abbiano scarso impatto sull'architettura del sistema. Infine, nella fase di transizione i requisiti dovrebbero essere definitivamente stabilizzati e le attività di progetto e implementazione dovrebbero essere limitate alla correzione degli ultimi errori.

Questo andamento è schematizzato in fig. 3.11. Le linee tratteggiate delimitano le iterazioni (che non necessariamente sono di durata uniforme come sembra mostrare la figura) e le cinque curve mostrano, in modo molto qualitativo e del tutto ipotetico, l'impegno richiesto dalle attività nel corso del processo di sviluppo.

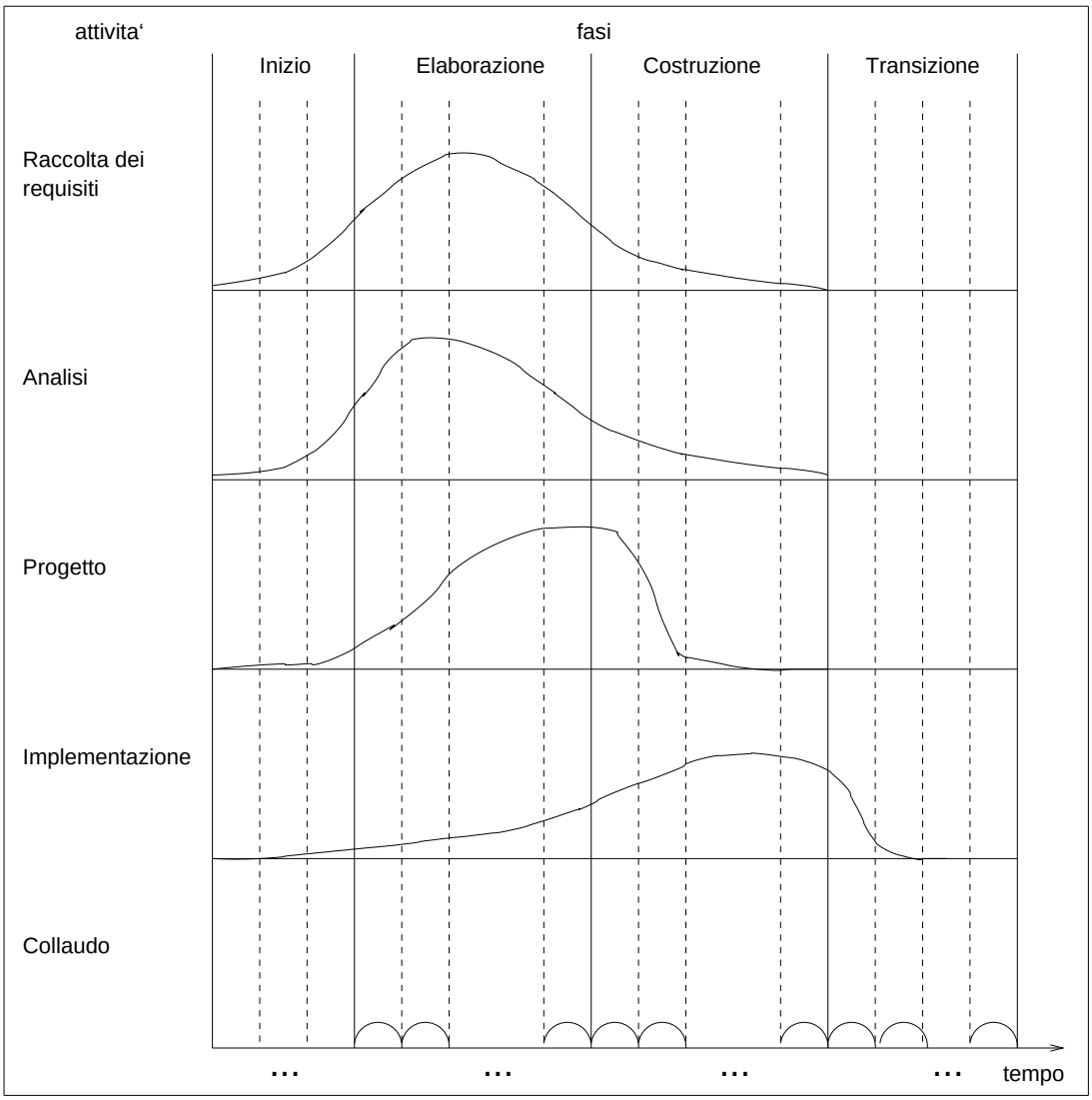


Figura 3.11: Fasi e attività nello Unified Process.

3.3.3 Processi agili

I processi agili sono una famiglia di processi evolutivi caratterizzati da iterazioni veloci, enfasi sulla produzione e collaudo di codice, e interazione col committente/utente [31]. Il materiale in questa sezione è tratto da un seminario[14] presentato dall'Ing. Riccardo Viviani della Imagicle nell'aa. 2018-19.

[14] `http://www.ing.unipi.it/~a009435/issw/extra/viviani.pdf`

Princípi generali

Gli autori dei primi processi agili pubblicarono il *Manifesto agile*[15] che ne riassume le motivazioni:

Persone e interazioni	sono più importanti dei processi e degli strumenti;
Un software funzionante	è più importante della documentazione;
Collaborare con i clienti	è più importante del contratto;
Aderire ai cambiamenti	è più importante che aderire al progetto.

Ovvero, fermo restando il valore (e quindi la necessità) delle entità a destra, consideriamo più importanti le entità a sinistra.

Da queste affermazioni derivano i seguenti princípi:

- soddisfazione del cliente;
- accogliere favorevolmente i cambiamenti anche se si è avanti con lo sviluppo;
- usare scale temporali brevi (feedback piú rapido possibile);
- eccellenza tecnica e buon design;
- team che si autoregolano e persone motivate;
- lavorare insieme quotidianamente (giocare per vincere);
- semplicità del progetto e software funzionante;
- iterazioni regolari e valutazione dell'attività;
- fornire a chi lavora al progetto l'ambiente, il supporto e la fiducia necessari al completamento del lavoro.

Su questi princípi si sono sviluppate numerose metodologie agili, fra cui Extreme Programming (XP), SCRUM, CRYSTAL e molte altre.

Una motivazione importante per l'adozione di tecnologie agili è la rapidità con cui permettono di scoprire difetti nel prodotto e nei semilavorati. La fig. 3.12 mostra qualitativamente la relazione fra i costi della ricerca ed eliminazione dei difetti e il ritardo, rispetto all'inizio del processo, con cui i difetti vengono scoperti. Sul grafico sono messi in evidenza i punti in cui certi tipi di difetto vengono scoperti usando tecniche agili (sottolineate in figura) e tecniche tradizionali (non sottolineate). Si può vedere come le tecniche agili permettano di intervenire piú tempestivamente sui difetti.

Il processo *Agicle*

Il processo Agicle (fig. 3.13) si basa sulle metodologie *Scrum* ed *XP*. Dopo una breve fase iniziale, si articola in una serie di cicli chiamati *sprint*, ciascuno dei quali può durare due settimane e produce un *incremento*.

I *partecipanti* al progetto si dividono in:

- *stakeholder*: rappresentanti del committente;

[15] `http://agilemanifesto.org`

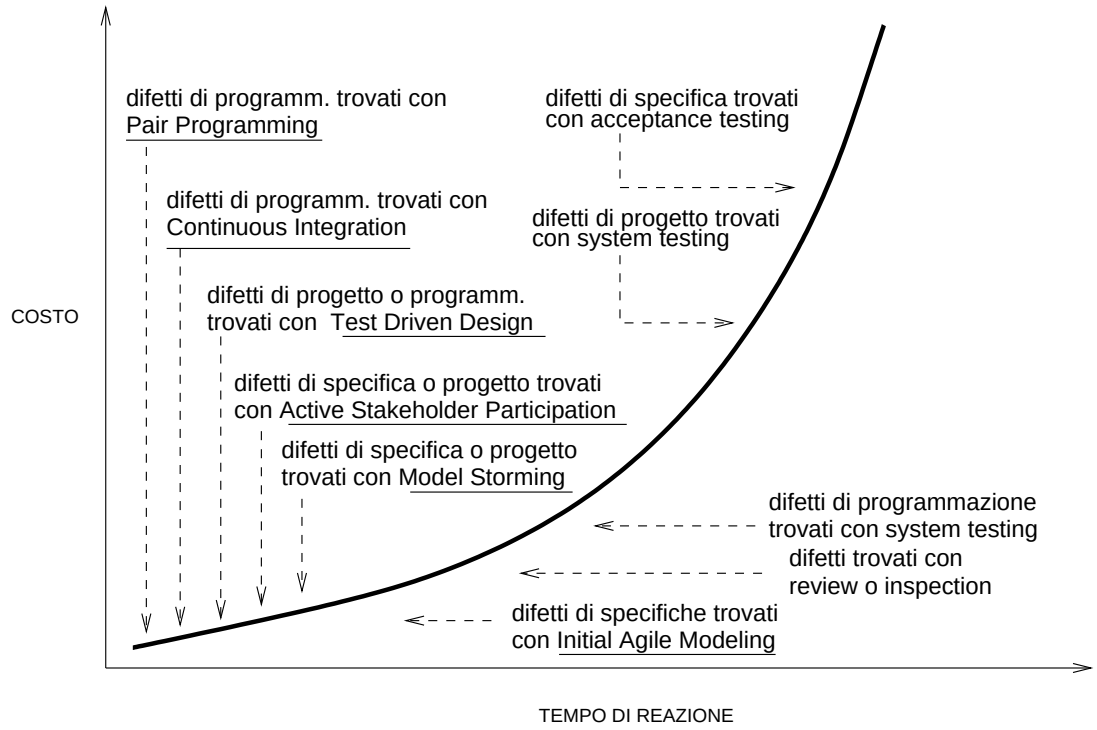


Figura 3.12: Costo dei difetti in funzione del tempo di identificazione.

- *product owner*: rappresentante del committente che *partecipa a tutte le attività insieme agli sviluppatori*;
- *scrum master*: facilitatore delle comunicazioni nel team e fra team e stakeholder, *partecipa alle attività di valutazione dello sprint.*

Nella fase iniziale, gli stakeholder raccolgono le *user story*, documenti simili, ma non uguali, a requisiti o casi d'uso. Le user story descrivono una funzione o caratteristica del sistema e devono avere le seguenti proprietà (*INVEST*):

independent: indipendenti l'una dall'altra, per quanto possibile.
negotiable: negoziabili, possono essere aggiunte, rimosse, modificate, unite, scomposte.
valuable: rilevanti, devono avere valore per l'utente, l'utente in genere non è il committente.
estimable: misurabili, è importante poterne stabilire la dimensione in termini di sforzo.
small: piccole, abbastanza da poter essere implementate in un tempo fra uno e dieci giorni.
testable: collaudabili, devono essere forniti i criteri di accettazione di ogni storia.

Ad ogni user story si assegna un *business value* che misura la sua importanza dal punto di vista degli stakeholder, ed uno *story point* che è una stima della complessità della sua implementazione. Inoltre nella user story si índicano il ruolo dello stakeholder che la propone, il suo obiettivo ed il suo scopo, come nel seguente esempio:

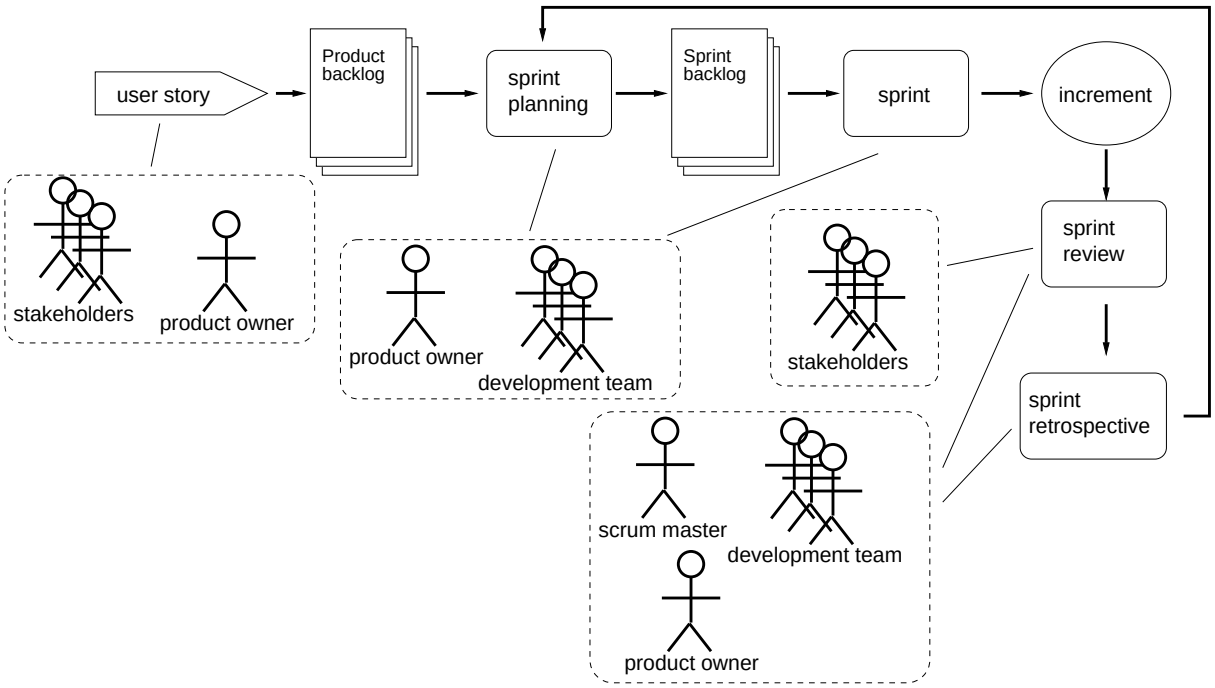


Figura 3.13: Il processo *Agicle*

come:	utente
voglio	che ad ogni tipo di file sia associata una specifica icona
al fine di	riconoscere il tipo di file in modo veloce senza necessità di aprirlo
business value	50
story point	2

Le user story vengono raccolte in un artefatto chiamato *product backlog* che è il semilavorato iniziale del processo e nel suo corso verrà rielaborato aggiungendo, togliendo e modificando user story, ed aggiungendo e togliendo difetti da correggere. Una volta compilato il product backlog iniziale, comincia la sequenza degli sprint. Ogni sprint si articola cosí:

sprint planning: scelta degli obiettivi da realizzare con l'incremento, estratti dal product backlog ed inseriti nello *sprint backlog*.
sprint (development): realizzazione dell'incremento.
sprint review: presentazione dell'incremento agli stakeholder.
sprint retrospective: discussione dello sprint concluso per preparare lo sprint successivo.

3.3.4 Modelli trasformazionali

Nei modelli trasformazionali, il processo di sviluppo consiste in una serie di *trasformazioni* che, partendo da una specifica iniziale ad alto livello, producono delle descrizioni del sistema sempre piú dettagliate, fino al punto in cui si può passare alla fase di codifica.

Tutte le specifiche sono *formali*, cioè espresse in un linguaggio di tipo matematico, in modo che la correttezza di ciascuna versione delle specifiche rispetto alla versione precedente possa essere verificata in modo rigoroso (esclusa, naturalmente, la specifica iniziale, che pur essendo espressa formalmente non può essere verificata rispetto ai requisiti dell'utente, necessariamente informali). Le trasformazioni da una specifica all'altra sono anch'esse formali, e quindi tali da preservare la correttezza delle specifiche.

Un processo trasformazionale è analogo alla soluzione di un'equazione:

$$\begin{aligned} f(x,y) &= g(x,y) \\ \ldots &= \ldots \\ y &= h(x) \end{aligned}$$

La formula iniziale viene trasformata attraverso diversi passaggi, usando le regole dell'algebra, fino alla forma finale.

Questa analogia, però, non è perfetta, perché nel caso delle equazioni la soluzione contiene la stesse informazioni dell'equazione iniziale (in forma esplicita invece che implicita), mentre nel processo di sviluppo del software vengono introdotte nuove informazioni nei passi intermedi, man mano che i requisiti iniziali vengono precisati o ampliati.

Esempio

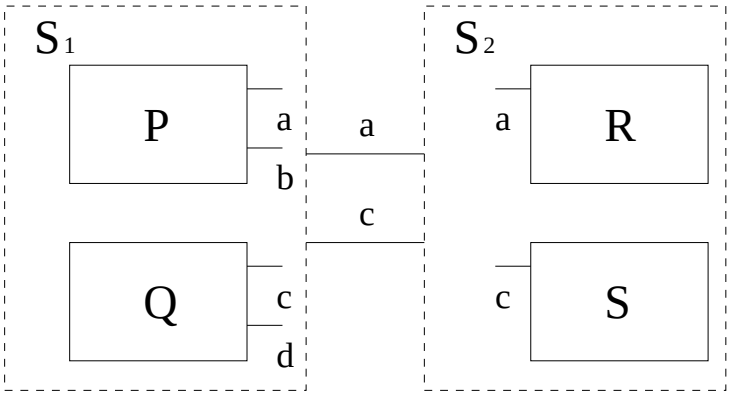


Figura 3.14: Struttura risultante dalla specifica iniziale.

Consideriamo, per esempio, un sistema di controllo contenente due sottosistemi, il sottosistema di acquisizione dati $\mathcal{S}_1$, formato dai due processi P e Q che controllano due sensori, e il sottosistema di interfaccia, $\mathcal{S}_2$, formato dai processi R ed S che devono visualizzare i dati su due schermi. Il processo P può eseguire le azioni a (comunicazione di dati) e b (attivazione di un allarme), il processo Q le azioni c (comunicazione di dati) e d (attivazione di un allarme), il processo R l'azione a, e il processo S l'azione c.

I due sensori sono indipendenti fra di loro e cosí i due schermi, quindi nel sottosistema $\mathcal{S}_1$ i processi P e Q si possono evolvere in modo concorrente senza alcun vincolo di sincronizzazione reciproca, cosí come i processi R ed S nel sottosistema $\mathcal{S}_2$.

I due sistemi $\mathcal{S}_1$ ed $\mathcal{S}_2$, invece, si devono scambiare informazioni attraverso le azioni a e c, e quindi si devono sincronizzare in modo da eseguire contemporaneamente l'azione a oppure l'azione c. La fig. 3.14 schematizza questa descrizione.

Un sistema di questo tipo può essere descritto con un formalismo appartenente alla famiglia delle *algebre dei processi*, per esempio col linguaggio LOTOS [10, 6]. In questo linguaggio sono definiti numerosi operatori di composizione fra processi, fra cui quello di *interleaving*, cioè l'esecuzione concorrente senza vincoli, rappresentato da $|||$, e quello di *sincronizzazione*, rappresentato da $|\,[\cdots]\,|$. A ciascun operatore sono associate una *semantica* che definisce il risultato della composizione (in termini delle possibili sequenze di azioni eseguite dal processo risultante) e delle *regole di trasformazione* (o *di inferenza*) sulle espressioni contenenti l'operatore.

Il sistema considerato viene quindi rappresentato dalla seguente espressione LOTOS:

$$(P[a,b] \;|||\; Q[c,d]) \;|[a,c]|\; (R[a] \;|||\; S[c])$$

L'espressione si può leggere cosí:

> I processi P e Q possono eseguire qualsiasi sequenza di azioni degli insiemi $\{a,b\}$ e $\{c,d\}$, rispettivamente, senza vincoli reciproci, e analogamente i processi R ed S con le azioni degli insiemi $\{a\}$ e $\{c\}$. I due sistemi $(P[a,b] \;|||\; Q[c,d])$ e $(R[a] \;|||\; S[c])$ sono sincronizzati su a e c, cioè devono eseguire contemporaneamente ciascuna di queste azioni.

Nel linguaggio LOTOS l'operatore di interleaving e quello di sincronizzazione sono definiti in modo tale che si possano manipolare analogamente agli operatori aritmetici di somma e di moltiplicazione, per cui l'espressione precedente è formalmente analoga a questa:

$$(P + Q) \cdot (R + S)$$

da cui

$$P \cdot R + P \cdot S + Q \cdot R + Q \cdot S \;.$$

Poiché P ed S (Q ed R) non hanno azioni in comune, il processo risultante dalla loro sincronizzazione non può generare alcuna azione e si può eliminare, ottenendo l'espressione

$$P \cdot R + Q \cdot S \;,$$

corrispondente a

$$(P[a,b] \;|[a]|\; R[a]) \;|||\; (Q[c,d] \;|[c]|\; S[c]) \;.$$

La nuova espressione si può leggere cosí:

> I processi P ed R sono sincronizzati su a, I processi Q ed S sono sincronizzati su c. I due sistemi $(P[a,b] \;|[a]|\; R[a])$ e $(Q[c,d] \;|[c]|\; S[c])$ possono eseguire qualsiasi sequenza permessa dai rispettivi vincoli di sincronizzazione interni, senza vincoli reciproci.

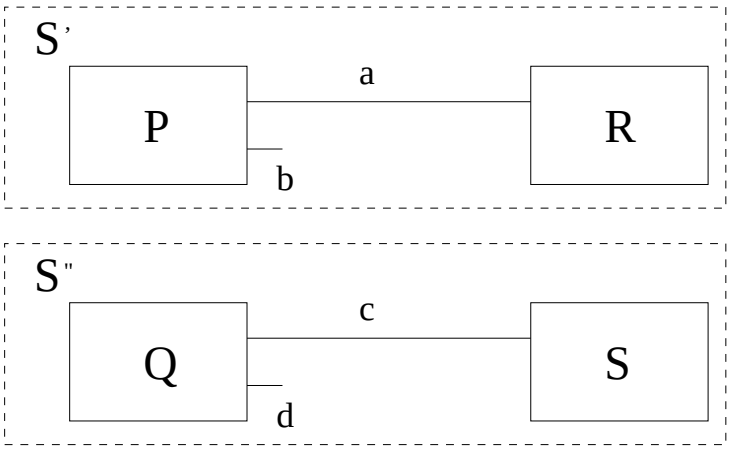


Figura 3.15: Struttura trasformata.

La seconda espressione è semanticamente equivalente alla prima poiché descrive lo stesso insieme di possibili sequenze di azioni descritto dall'espressione originale, ma rappresenta una diversa organizzazione interna (e quindi una possibile implementazione) del sistema, che adesso è scomposto nei due sottosistemi $\mathcal{S}' = \{P, R\}$ e $\mathcal{S}'' = \{Q, S\}$ (fig. 3.15). Il nuovo raggruppamento dei processi è presumibilmente migliore del precedente, perché mette insieme i processi che devono interagire e separa quelli reciprocamente indipendenti.

Letture

Obbligatorie: cap. 7 Ghezzi, Jazayeri, Mandrioli [18], oppure cap. 1 Ghezzi et al. [16], oppure cap. 3 Pressman [40].

Facoltative: sez. 13.7.2 Pressman [40], cap. 2 Arlow, Neustadt [8].

Capitolo 4

Analisi e specifica dei requisiti

In questo capitolo presentiamo alcuni linguaggi e metodi usati nella fase di analisi e specifica dei requisiti. I requisiti descrivono ciò che l'utente si aspetta dal sistema, e *specificarli* significa esprimerli in modo chiaro, univoco, consistente e completo. I requisiti si distinguono in *funzionali* e *non funzionali*. In questo capitolo, il termine *requisiti* si riferisce ai comportamenti e proprietà del sistema come espressi nei documenti di specifica.

4.1 I requisiti

I requisiti funzionali descrivono cosa deve fare il sistema, generalmente in termini di relazioni fra dati di ingresso e dati di uscita, oppure fra *stimoli* (dall'ambiente al sistema) e *risposte* del sistema. Questi requisiti sono generalmente esprimibili in modo formale.

I requisiti non funzionali esprimono dei *vincoli* o delle caratteristiche di qualità. Queste ultime sono piú difficili da esprimere in modo formale, in particolare è difficile esprimerle in modo quantitativo. Fra le caratteristiche di qualità del software ricordiamo le seguenti:

sicurezza (safety): capacità di funzionare senza arrecare danni a persone o cose (piú precisamente, con un rischio limitato a livelli accettabili). Si usa in questo senso anche il termine *innocuità*.

riservatezza (security): capacità di impedire accessi non autorizzati ad un sistema, e in particolare alle informazioni in esso contenute. Spesso il termine *sicurezza* viene usato anche in questo senso[1].

robustezza: capacità di funzionare in modo accettabile anche in situazioni non previste, come guasti o dati di ingresso errati.

prestazioni: uso efficiente delle risorse, come il tempo di esecuzione e la memoria centrale.

disponibilità: capacità di rendere disponibile un servizio continuo per lunghi periodi.

usabilità: facilità d'uso.

interoperabilità: capacità di integrazione con altre applicazioni.

[1] L'autore di questa dispensa sconsiglia l'uso del termine *cybersicurezza*, semmai *ciber*sicurezza.

I requisiti di sicurezza, riservatezza e robustezza sono aspetti del piú generale requisito di *affidabilità* (*dependability*), nel senso piú corrente di questo termine. Ricordiamo che le definizioni piú rigorose di questo termine (corrispondenti al termine inglese *reliability*) si riferiscono alla probabilità che non avvengano malfunzionamenti entro determinati periodi di tempo.

4.1.1 Tracciabilità e collaudabilità

Come già accennato, la tracciabilità è la possibilità di ricostruire i nessi logici fra requisiti a diverso livello di astrazione, fra requisiti e scelte di progetto, fra scelte di progetto e scelte di codifica. Tutte queste informazioni devono poi essere riconducibili ai test. Questa rete di connessioni logiche deve poter essere percorsa anche in senso inverso, per cui ogni caso di test deve servire a verificare qualche aspetto del codice o del progetto, e infine la conformità a qualche requisito.

La tracciabilità richiede che la documentazione prodotta nel processo di sviluppo contenga queste informazioni esplicitamente, e che tali informazioni vengano aggiornate, e soprattutto riesaminate, ogni volta che si facciano dei cambiamenti. Se, per esempio, si modifica il codice di un sottoprogramma, bisogna fare un tracciamento "all'indietro", cioè verso i concetti piú astratti, per assicurarsi che le modifiche siano compatibili con le scelte di progetto e con le specifiche. Bisogna fare anche il tracciamento "in avanti", cioè verso i concetti piú concreti, per assicurasi che il sistema modificato continui a passare i test che passava prima (test *di regressione*) oppure progettare nuovi casi di test adatti alla nuova versione del codice. Se si scoprisse un conflitto fra le modifiche e il progetto o le specifiche, bisogna analizzare criticamente tutti e tre i tipi di informazioni: se le modifiche nel codice sono motivate da qualche errore o lacuna nel progetto o nelle specifiche, si cambieranno quelle parti in modo da mantenere la coerenza fra i vari modelli del sistema.

Osserviamo che una possibile discrepanza fra specifiche da una parte e progetto e codifica dall'altra è l'introduzione di *funzioni non richieste* (*unintended functions*). È possibile, cioè, che certi cambiamenti a livello di progetto o di codice permettano al sistema di fare cose non richieste, introducendo il rischio di comportamenti imprevisti e presumibilmente dannosi, oltre che difficili da rilevare anche con estese campagne di collaudo.

Quanto sopra mostra l'importanza della *collaudabilità* dei requisiti e delle scelte di progetto: la documentazione deve esprimere ogni requisito o scelta di progetto in modo che la correttezza della sua implementazione si possa verificare con dei test. Questo vale, in particolare, per i requisiti funzionali e non funzionali. Dare definizioni collaudabili di questi ultimi può essere difficile, perché spesso non hanno un preciso significato quantitativo. In questi casi si possono definire delle metriche empiriche: per esempio, l'usabilità di un'applicazione si può valutare in base al tempo medio impiegato da un gruppo di utenti (alfa o beta tester) per imparare ad eseguire certe operazioni.

4.2 Classificazioni dei sistemi software

Nell'affrontare l'analisi dei requisiti, è utile individuare certe caratteristiche generali del sistema che dobbiamo sviluppare. A questo scopo possiamo considerare alcuni criteri di classificazione dei sistemi, esposti nel resto di questa sezione.

4.2.1 Requisiti temporali

Una prima importante classificazione delle applicazioni può essere fatta in base ai requisiti temporali, rispetto ai quali i sistemi si possono caratterizzare come:

sequenziali: senza vincoli di tempo.
concorrenti: con sincronizzazione fra processi.
in tempo reale: con tempi di risposta prefissati.

Nei sistemi sequenziali un risultato corretto (rispetto alla specifica del sistema in termini di relazioni fra ingresso e uscita) è accettabile qualunque sia il tempo impiegato per ottenerlo. Naturalmente è sempre desiderabile che l'elaborazione avvenga velocemente, ma la tempestività del risultato non è un requisito funzionale, bensí un requisito relativo alle prestazioni o all'usabilità. Inoltre, un sistema è sequenziale quando è visto come un singolo processo, le cui interazioni con l'ambiente (operazioni di ingresso e di uscita) avvengono in una sequenza prefissata.

I sistemi concorrenti, invece, sono visti come insiemi di processi autonomi (eventualmente *distribuiti*, cioè eseguiti da piú di un elaboratore) che in alcuni momenti possono comunicare fra di loro. Le interazioni reciproche dei processi, e fra questi e l'ambiente, sono soggette a vincoli di sincronizzazione ed avvengono in sequenze non determinate a priori. Per *vincoli di sincronizzazione* si intendono delle relazioni di precedenza fra eventi, come, per esempio, "*l'azione* a *del processo* P *deve essere eseguita dopo l'azione* b *del processo* Q", oppure "*la valvola n. 2 non si deve aprire prima che si sia chiusa la valvola n. 1*". In generale, un insieme di vincoli di sincronizzazione su un insieme di processi interagenti può essere soddisfatto da diverse sequenze di eventi, ma il verificarsi di sequenze che violano tali vincoli è un malfunzionamento.

Un sistema in tempo reale è generalmente un sistema concorrente, e in piú deve fornire i risultati richiesti entro limiti di tempo prefissati: in questi sistemi un risultato, anche se corretto, è inaccettabile se non viene prodotto in tempo utile. Per esempio, una coppia produttore/consumatore senza controllo di flusso, in cui il produttore funziona ad un ritmo indipendente da quello del consumatore, è un sistema real-time, poiché in questo caso il produttore impone al consumatore un limite massimo sul tempo di esecuzione. Se il consumatore non rispetta questo limite si perdono delle informazioni. Un esempio di questo tipo di sistema è un sensore che manda informazioni a un elaboratore, dove la frequenza di produzione dei dati dipende solo dal sensore o dall'evoluzione del sistema fisico controllato.

4.2.2 Tipo di elaborazione

Un'altra classificazione dei sistemi si basa sul tipo di elaborazione compiuta prevalentemente. I sistemi si possono quindi caratterizzare come:

orientati ai dati: mantengono e rendono accessibili grandi quantità di informazioni (p.es., banche dati, applicazioni gestionali).

orientati alle funzioni: trasformano informazioni mediante elaborazioni complesse (p.es., compilatori).

orientati al controllo: interagiscono con l'ambiente, modificando il proprio stato in séguito agli stimoli esterni (p.es., sistemi operativi, controllo di processi).

Bisogna però ricordare che ogni applicazione usa dei dati, svolge delle elaborazioni, ed ha uno stato che si evolve, in modo piú o meno semplice. Nello specificare un sistema è quindi necessario, in genere, prendere in considerazione tutti questi tre aspetti.

4.2.3 Software di base o applicativo

Un'altra classificazione si può ottenere considerando se il software da realizzare serve a fornire i servizi base di elaborazione ad altro software (per esempio, se si deve realizzare un sistema operativo o una sua parte), oppure software intermedio fra il software di base e le applicazioni (librerie), oppure software applicativo vero e proprio.

4.3 Linguaggi di specifica

Normalmente i requisiti, sia funzionali che non funzionali, vengono espressi in linguaggio naturale. Questo, però, spesso non basta a specificare i requisiti con sufficiente precisione, chiarezza e concisione. Per questo sono stati introdotti numerosi *linguaggi di specifica* che possano supplire alle mancanze del linguaggio naturale.

Un problema fondamentale nell'uso del linguaggio naturale è la sua *ambiguità*, cioè il fatto che certe frasi si possono interpretare in modi diversi. Consideriamo questi classici esempi:

ogni uomo ama una donna. Si può interpretare in quattro modi:

- per ogni uomo x esiste almeno una donna y tale che x ama y;
- per ogni uomo x esiste una e una sola donna y tale che x ama y;
- esiste almeno una donna y tale che ogni uomo ama y;
- esiste una e una sola donna y tale che ogni uomo ama y.

time flies like an arrow. Si può interpretare in molti modi a seconda dell'interpretazione delle parole, fra cui:

- *time* = tempo, *flies* = vola, *like* = come: Il tempo vola come una freccia;
- *time* = temporali, *flies* = mosche, *like* = piacere: alle mosche temporali piace una freccia;
- *time* = cronometrare, *flies* = mosche, *like* = come: cronometra le mosche (veloce) come una freccia;
- *time* = cronometrare, *flies* = mosche, *like* = simili: cronometra le mosche simili a una freccia;
- ...

ho visto Maria nel bosco con gli occhiali. Si può interpretare in molti modi a seconda della parola a cui si riferisce un'altra:

- ho visto Maria che stava nel bosco e aveva gli occhiali;
- avevo gli occhiali e ho visto Maria che stava nel bosco;
- stavo nel bosco e ho visto Maria che aveva gli occhiali;
- stavo nel bosco, avevo gli occhiali e ho visto Maria;
- ...

I verbi modali (come *volere* e *potere*) sono spesso causa di ambiguità:

- *X non deve fare Y*: *Y* è facoltativo o è vietato?
- *X non può fare Y*: *Y* è materialmente impossibile o è vietato?

I problemi del linguaggio naturale si possono mitigare introducendo regole di scrittura restrittive, in particolare definendo chiaramente il significato di certi termini escludendone interpretazioni alternative. Ancor meglio, conviene usare il piú possibile dei linguaggi formali, che verranno trattati in séguito.

4.3.1 Classificazione dei formalismi di specifica

I formalismi usati nella specifica dei sistemi privilegiano in grado diverso gli aspetti dei sistemi considerati piú sopra. Alcuni formalismi sono specializzati per descrivere un aspetto particolare, come la specifica dei dati o delle funzioni, mentre altri si propongono una maggiore generalità. Alcune metodologie di sviluppo si affidano ad un unico formalismo, mentre altre ne sfruttano piú d'uno.

I formalismi per la specifica, quindi, possono essere suddivisi analogamente ai tipi di sistemi per cui sono concepiti, per cui si avranno formalismi orientati ai dati, alle funzioni, e via dicendo. Inoltre, i formalismi di specifica vengono classificati anche secondo i due criteri del *grado di formalità* e dello *stile di rappresentazione*, che descriviamo di séguito.

Grado di formalità

I linguaggi si possono suddividere in *formali*, *semiformali*, *informali*.

Un linguaggio è *formale* se la sua sintassi e la sua semantica sono definite in modo matematicamente rigoroso; il significato di questa frase sarà meglio definito nella parte dedicata alla logica, dove vedremo come la sintassi e la semantica di un linguaggio si possano esprimere per mezzo di concetti matematici elementari, come insiemi, funzioni e relazioni.

Una specifica espressa in un linguaggio formale è precisa e verificabile, e inoltre lo sforzo di traduzione dei concetti dal linguaggio naturale a un linguaggio formale aiuta la comprensione di tali concetti da parte degli analisti. Questo è dovuto al fatto che, dovendo riformulare in un linguaggio matematico un concetto espresso in linguaggio naturale, si è costretti ad eliminare le ambiguità e ad esplicitare tutte le ipotesi date per scontate nella forma originale.

Il maggior limite dei linguaggi formali è il fatto che richiedono un certo sforzo di apprendimento e sono generalmente poco comprensibili per chi non abbia una preparazione adeguata.

Per notazioni o linguaggi *semiformali* si intendono quelle che hanno una sintassi (spesso grafica) definita in modo chiaro e non ambiguo ma non definiscono una semantica per mezzo di concetti matematici, per cui il significato dei simboli usati viene espresso in modo informale. Nonostante questo limite, i linguaggi semiformali sono molto usati perché permettono di esprimere i concetti in modo piú conciso e preciso del linguaggio naturale, e sono generalmente piú facili da imparare ed usare dei linguaggi formali.

I linguaggi *informali* non hanno né una sintassi né una semantica definite rigorosamente. I linguaggi naturali hanno una sintassi codificata dalle rispettive grammatiche, ma non formalizzata matematicamente, ed una semantica troppo ricca e complessa per essere formalizzata. Al di fuori del linguaggio naturale non esistono dei veri e propri linguaggi informali, ma solo delle notazioni grafiche inventate ed usate liberamente per schematizzare qualche aspetto di un sistema, la cui interpretazione viene affidata all'intúito del lettore ed a spiegazioni in linguaggio naturale. I disegni fatti alla lavagna durante le lezioni rientrano in questo tipo di notazioni.

Stile di rappresentazione

Un'altra possibile suddivisione è fra linguaggi *descrittivi* e *operazionali*.

La differenza fra questi linguaggi può essere compresa considerando, per esempio, un semplice sistema costituito da un contenitore di gas con una valvola di scarico. I requisiti di sicurezza esigono che quando la pressione p del gas supera un certo valore di soglia P la valvola si apra, e quando la pressione torna al di sotto del valore di soglia la valvola si chiuda. Possiamo esprimere questo requisito con una formula logica:

$$p < P \Leftrightarrow \textbf{valvola-chiusa}$$

in cui **valvola-chiusa** è una proposizione che è vera quando la valvola è chiusa. In questo caso, il sistema è rappresentato da una relazione logica fra le proprietà delle entità coinvolte, e si ha quindi una rappresentazione descrittiva (o *dichiarativa*).

Lo stesso sistema può avere una rappresentazione operazionale, cioè in termini di una *macchina astratta*, che si può trovare in certo *stato* e passare ad un altro stato (effettuando cioè una *transizione*) quando avvengono certi *eventi*. La figura 4.1 mostra la macchina astratta corrispondente all'esempio, che si può trovare nello stato "aperto" o "chiuso" secondo il valore della pressione. Le espressioni 'when($p \geq P$)' e 'when($p < P$)' denotano gli eventi associati al passaggio della pressione a valori, rispettivamente, maggiori o minori della soglia P (la prima espressione, per esempio, significa "$p \geq P$ diventa vero").

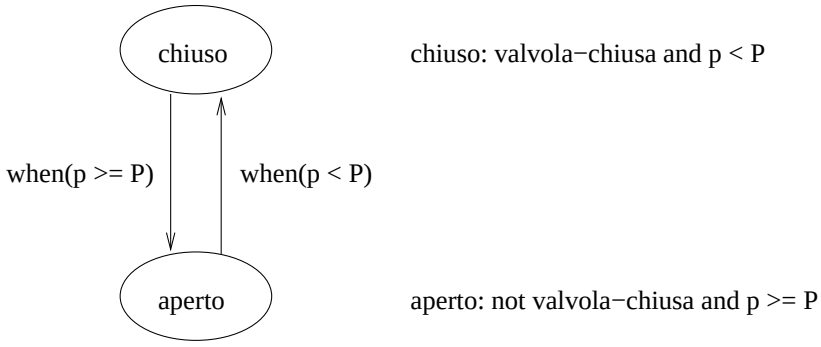


Figura 4.1: Una descrizione operazionale, e sua corrispondenza con la descrizione dichiarativa.

Quindi i linguaggi descrittivi rappresentano un sistema in termini di entità costituenti, delle loro proprietà, e delle relazioni fra entità, mentre i linguaggi operazionali lo rappresentano in termini di stati e transizioni che ne definiscono il comportamento. Spesso una specifica completa richiede che vengano usate tutte e due le rappresentazioni.

Osserviamo che, nel nostro esempio, la formulazione descrittiva esprime direttamente il requisito di sicurezza, mentre la formulazione operazionale descrive un comportamento del sistema tale che rispetti il requisito, ed è quindi meno astratta.

4.4 Formalismi orientati al controllo

Questi formalismi descrivono gli aspetti dei sistemi relativi alla loro evoluzione temporale, alle possibili sequenze di eventi o di azioni, alla sincronizzazione fra le attività dei loro sottosistemi, o fra il sistema e l'ambiente in cui opera. Questi aspetti sono particolarmente importanti nei sistemi *reattivi*, che devono reagire a stimoli provenienti dall'ambiente, che si presentano in un ordine generalmente non prevedibile.

I formalismi orientati al controllo sono un campo di studio molto vasto e articolato. In questo corso verranno date solo alcune nozioni elementari relative al formalismo degli *automi a stati finiti* ed alle sue estensioni adottate nel linguaggio UML (sez. 4.6.7).

4.4.1 Automi a stati finiti

Col formalismo degli *automi a stati finiti* (ASF) o *macchine a stati*[2] descriviamo un sistema attraverso gli *stati* in cui si può trovare, le *transizioni*, cioè i passaggi da uno stato all'altro, gli *ingressi* (stimoli esterni) che causano le transizioni, le *uscite* del sistema, che possono essere associate alle transizioni (*macchine di Mealy*) o agli stati (*macchine di Moore*), e lo *stato iniziale* da cui parte l'evoluzione del sistema.

Osservazione. Nel campo dei compilatori, la sintassi dei linguaggi da compilare viene specificata per mezzo di ASF i cui ingressi corrispondono alla lettura di simboli. In base ai caratteri letti, gli ASF eseguono transizioni fino ad arrivare ad uno stato in cui si è *riconosciuta* una particolare struttura sintattica (sez. 6.3.4).

L'ambiente esterno agisce sul sistema rappresentato dall'automa generando una successione di ingressi discreti, e il sistema risponde a ciascun ingresso cambiando il proprio stato (eventualmente restando nello stato corrente) e generando un'uscita. L'automa definisce le regole secondo cui il sistema risponde agli stimoli esterni.

Un ASF è quindi definito da una sestupla

$$< S, I, U, d, t, s_0 >$$

con

- S: insieme degli stati;
- I: insieme degli ingressi;
- U: insieme delle uscite;
- $d : S \times I \to S$: funzione di transizione;
- $t : S \times I \to U$: funzione di uscita (macchine di Mealy);
- $s_0 \in S$: stato iniziale.

Nelle macchine di Moore si ha $t : S \to U$.

Un ASF definisce quindi un grafo orientato, i cui nodi sono gli stati, e gli archi orientati, etichettati dagli ingressi e dalle uscite, descrivono la funzione di transizione e la funzione di uscita. Useremo una notazione grafica in cui gli stati si rappresentano con cerchi o rettangoli ovalizzati. Lo stato iniziale viene indicato da un arco senza stato di origine.

L'automa rappresentato in fig. 4.2 descrive l'interazione fra un utente ed un centralino che accetta chiamate interne a numeri di due cifre e chiamate esterne a numeri di tre cifre, precedute dallo zero (esempio da [16]). Nel diagramma, le transizioni sono etichettate con espressioni della forma '*ingresso*/*uscita*'; alcune transizioni non producono uscite. Un ingresso della forma '*m*:*n*' rappresenta una cifra fra m e n. Ogni stato è identificato sia da un nome che da un numero. Al numero si farà riferimento nella rappresentazione tabulare (v. oltre) dell'automa.

La sestupla che definisce l'automa è quindi cosí costituita:

[2] Precisiamo che sono stati descritti diversi tipi di macchine a stati; nel séguito faremo riferimento a uno dei tipi usati piú comunemente, i *trasduttori deterministici*.

- S = {telefono riagganciato, attesa prima cifra, attesa prima cifra esterna, attesa seconda cifra esterna, attesa terza cifra esterna, attesa seconda cifra interna, attesa collegamento, colloquio impossibile, ricevente squilla, dialogo};
- I = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, sollevamento, sollevamento ricevente, riaggancio, occupato, libero};
- U = {segnale interno, segnale esterno, segnale occupato, segnale libero};
- d: vedi diagramma;
- t: vedi diagramma;
- s_0 = telefono riagganciato.

Non si distingue fra riaggancio del chiamante e del chiamato.

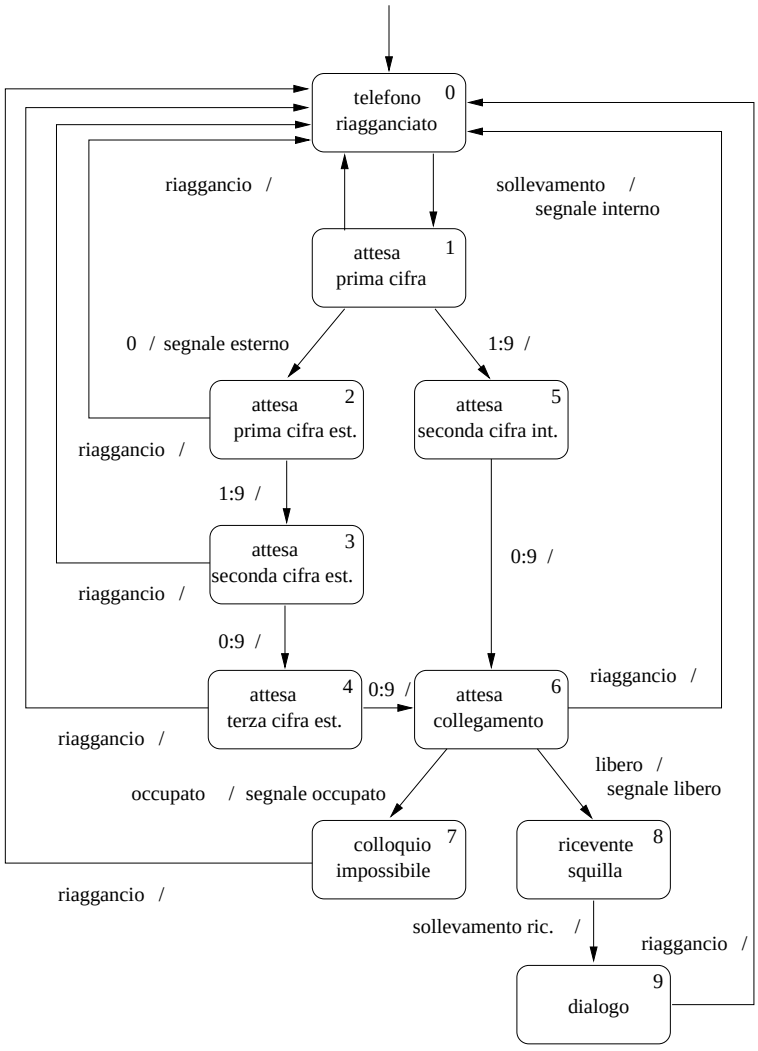


Figura 4.2: Un automa a stati finiti

Osservare che, per ogni stato, le transizioni uscenti sono mutuamente esclusive: l'automa è quindi *deterministico*, in quanto per ogni coppia (*stato*, *ingresso*) è possibile al piú una transizione.

Tabella 4.1: Rappresentazione tabulare

	0	1	2	3	4	5	6	7	8	9
0		S/int.								
1	R/		0/est.			1:9/				
2	R/			1:9/						
3	R/				0:9/					
4	R/						0:9/			
5	R/						0:9/			
6	R/							O/occ.	L/lib.	
7	R/									
8										Sr/
9	R/									

R: riaggancio; **S**: sollevamento del chiamante; **O**: ricevente occupato; **L**: ricevente libero; **Sr**: sollevamento del ricevente

Un automa può essere rappresentato anche per mezzo di tabelle. Una rappresentazione possibile si basa su una matrice quadrata di ordine n (numero degli stati): se esiste una transizione dallo stato s_i allo stato s_j, l'elemento della matrice sulla riga i e la colonna j contiene l'ingresso che causa la transizione e la corrispondente uscita. La rappresentazione tabulare dell'ASF di fig. 4.2 è data dalla tabella 4.1.

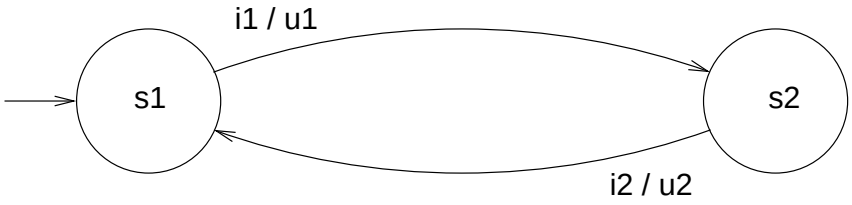


Figura 4.3: Un automa a stati finiti

Un ASF può essere definito anche in modo testuale, usando appositi linguaggi. Il semplice automa di fig. 4.3 può essere definito, per esempio, nel linguaggio SAL (*Symbolic Analysis Laboratory* [12]) come segue:

```
esempio: CONTEXT =
BEGIN
  TStato: TYPE = {s1, s2};              % insieme degli stati
  TIngresso: TYPE = {i1, i2};           % insieme degli ingressi
  TUscita: TYPE = {u1, u2};             % insieme delle uscite
  automa: MODULE =
  BEGIN
      INPUT input: TIngresso
      LOCAL stato: TStato
      OUTPUT uscita: TUscita
      INITIALIZATION stato = s1         % stato iniziale
```

```
      TRANSITION [
           % transizione s1 -> s2
           stato = s1 AND ingresso = i1  % guardia
           --> stato' = s2;              % stato successivo
               uscita' = u1              % uscita successiva
           % transizione s2 -> s1
         [] stato = s2 AND ingresso = i2
           --> stato' = s1;
               uscita' = u2
         [] ELSE --> stato' = stato
      ]
  END;
END
```

In questo esempio vediamo la definizione degli insiemi di stati, ingressi e uscite. Per ogni transizione, una *guardia*[3] dichiara lo stato di origine e l'ingresso che la abilita. La guardia è seguita da istruzioni che dichiarano quali sono lo stato successivo e l'uscita associata alla transizione, denotati da apici. Questa rappresentazione dell'automa corrisponde alla definizione formale data in questo capitolo, in cui lo stato è un concetto atomico. Il linguaggio SAL permette di definire, come vedremo in séguito (sez. 4.5.9), sistemi piú complessi, formati da automi interagenti basati su stati strutturati.

Componibilità e scalabilità

Nella specifica di sistemi complessi, e in particolare di sistemi concorrenti, è spesso necessario o conveniente rappresentare il sistema complessivo come un aggregato di sottosistemi. Un formalismo di specifica ha la proprietà della *componibilità* se permette di costruire la rappresentazione del sistema complessivo per mezzo di semplici operazioni di composizione sulle rappresentazioni dei sottosistemi. Per *scalabilità* si intende la capacità di rappresentare un sistema in modo tale che la complessità di tale rappresentazione sia dello stesso ordine di grandezza della somma delle complessità dei singoli sottosistemi.

Nel caso degli ASF la componibilità e la scalabilità sono limitate, come mostrerà l'esempio sviluppato nei paragrafi successivi.

La figura 4.4 mostra tre sottosistemi: un produttore, un magazzino ed un consumatore (esempio da [16]). Il produttore si trova inizialmente nello stato p_1 in cui è pronto a produrre qualcosa (per esempio, un messaggio o una richiesta di elaborazione), e con la produzione passa allo stato p_2 in cui attende di depositare il prodotto; col deposito ritorna allo stato iniziale. Il consumatore ha uno stato iniziale c_1 in cui è pronto a prelevare qualcosa, e col prelievo passa allo stato c_2 in cui attende che l'oggetto venga consumato. Il magazzino ha tre stati, in cui, rispettivamente, è vuoto (m_1), contiene un oggetto (m_2), e contiene due oggetti (m_3); il magazzino passa da uno stato all'altro in séguito alle operazioni di deposito e di prelievo.

[3] Il termine "guardia" nel linguaggio SAL ha un significato leggermente diverso da quello che incontreremo in altri formalismi.

In questo modello, gli ingressi *produzione*, *deposito*, *prelievo* e *consumo* rappresentano il completamento di attività eseguite dal sistema modellato. Questo non corrisponde all'idea intuitiva di "ingresso" di un sistema, ma descrive correttamente un sistema di controllo che riceve segnali dal sistema controllato.

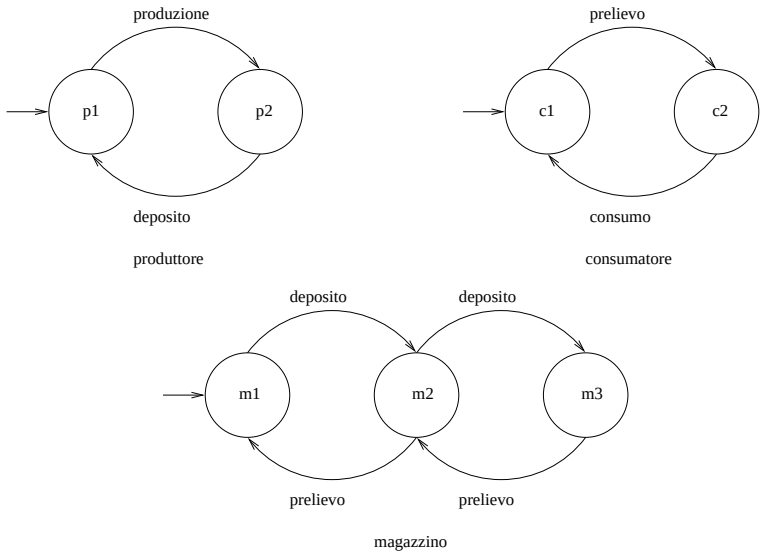


Figura 4.4: Esempio (1)

Il sistema complessivo, mostrato in figura 4.5, ha dodici stati, invece dei sette usati per descrivere i sottosistemi separatamente: ciascuno dei dodici stati è una possibile combinazione degli stati dei sottosistemi. In generale, componendo un numero n di sottosistemi in un sistema complessivo, si ha che:

1. l'insieme degli stati del sistema complessivo è il prodotto cartesiano degli insiemi degli stati dei sottosistemi;
2. cioè, ogni stato del sistema complessivo è una n-upla formata da stati dei sottosistemi, per cui viene nascosta la struttura gerarchica del sistema (gli stati dei sottosistemi vengono concentrati nello stato globale);
3. l'evoluzione del sistema viene descritta come se ad ogni passo uno solo dei sottosistemi potesse compiere una transizione, mentre in generale è possibile che transizioni in sottosistemi distinti possano avvenire in modo concorrente;
4. il numero degli stati del sistema totale cresce esponenzialmente col numero dei sottosistemi.

Mentre i punti 1 e 4 dimostrano la poca scalabilità degli ASF, il punto 3 ne mette in evidenza un'altra caratteristica: non esprimono la concorrenza dei sottosistemi, per cui si prestano solo alla specifica di sistemi sequenziali.

Il problema dell'aumento della complessità quando si compongono sottosistemi dipende dal fatto che, nel modello di ASF qui presentato, lo stato del sistema è *globale*, in quanto in un dato istante l'intero sistema viene modellato da un unico stato, e *atomico*,

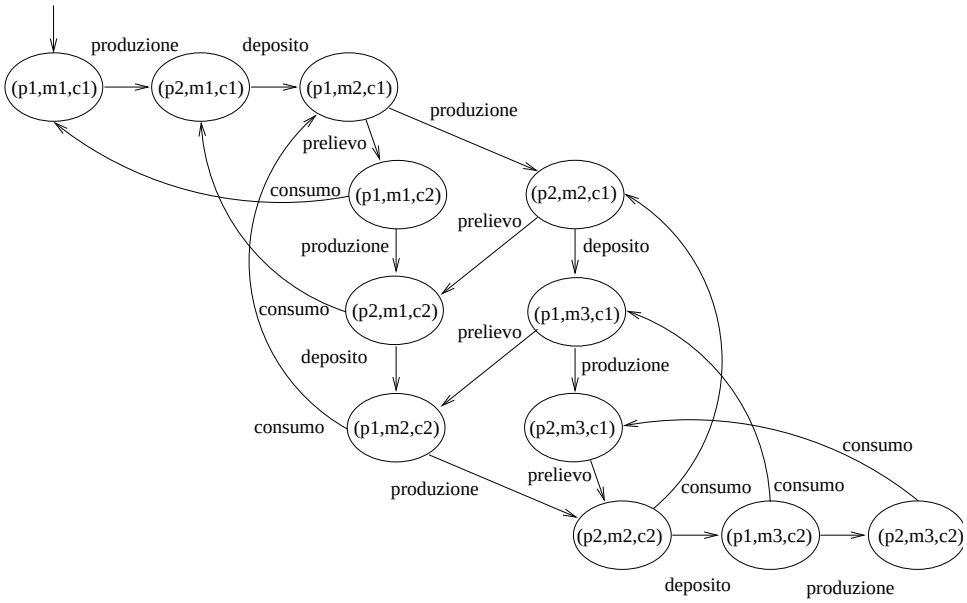


Figura 4.5: Esempio (2)

in quanto lo stato non contiene altra informazione che la propria identità e le transizioni e uscite definite dalle rispettive funzioni.

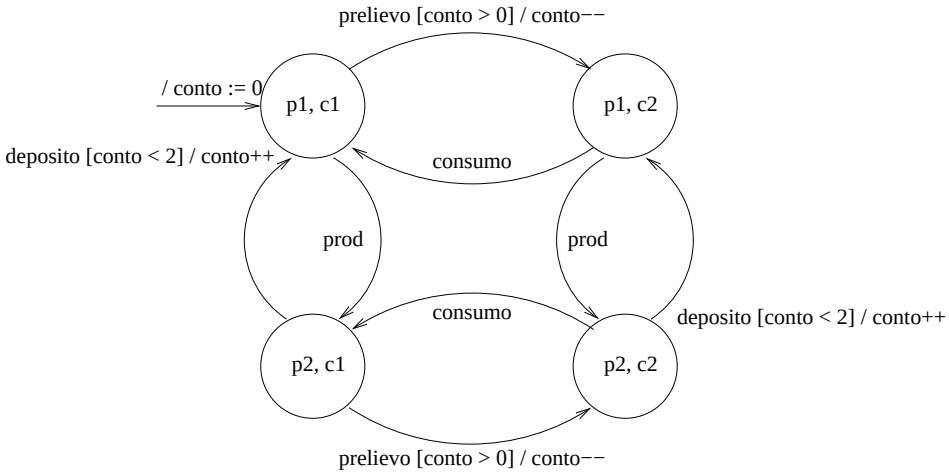


Figura 4.6: Esempio (3)

Un modo per rendere meno complesse le specifiche di sistemi per mezzo di ASF consiste nell'estendere il concetto di "stato" associandovi delle strutture dati. Nel nostro esempio, i tre stati dell'ASF che rappresenta il magazzino potrebbero essere sostituiti da un unico stato a cui è associata una variabile il cui valore è il numero di elementi immagazzinati. Le uscite associate alle operazioni di prelievo e di deposito sarebbero azioni di decremento e, rispettivamente, incremento della variabile (fig. 4.6). Il prelievo e il deposito, a loro volta, sarebbero condizionati dal valore della variabile, cosa che comporta l'estensione del concetto di "transizione", a cui si devono aggiungere delle condizioni (*guardie*) che devono essere soddisfatte affinché una transizione possa aver luogo.

Un'altra estensione degli ASF consiste nell'introduzione di stati composti, cioè descritti, a loro volta, da macchine a stati. In questo modo un sistema complesso può essere descritto ad alto livello da un automa con pochi stati, ciascuno di quali può essere decomposto in sottostati quando serve una specifica piú dettagliata. Questo metodo è alla base degli *Statechart*, un formalismo che esamineremo nel capitolo relativo ai metodi orientati agli oggetti.

Infine, nelle *reti di Petri* lo stato del sistema viene modellato in modo diverso, tale che la specifica renda esplicito e visibile il fatto che il sistema totale è composto da sottosistemi. Questo permette di descrivere sistemi concorrenti.

4.5 Logica

Was sich überhaupt sagen läßt, läßt sich klar sagen.
(Tutto ciò che può essere detto si può dire chiaramente).
— L. Wittgenstein, *Tractatus*, prefazione.

La logica serve a formalizzare il ragionamento, e in particolare a decidere in modo rigoroso (e quindi potenzialmente automatizzabile) se certe affermazioni sono vere e se dalla verità di certe affermazioni si può dedurre la verità di altre affermazioni. È evidente che la logica è fondamentale per qualsiasi forma di ragionamento scientifico, anche quando viene condotto per mezzo del linguaggio naturale, e in qualsiasi campo di applicazione, anche al di fuori delle discipline strettamente scientifiche e tecnologiche. In particolare, la logica moderna è stata sviluppata per servire da fondamento alle discipline matematiche, fra cui rientra gran parte della scienza dell'informazione.

Al di là del suo carattere fondamentale, la logica può essere usata come linguaggio di specifica. Nell'ingegneria del software, quindi, una logica serve da linguaggio di specifica, quando le proprietà di un sistema si esprimono in termini di formule logiche. Un sistema specificato per mezzo della logica può essere analizzato rigorosamente, e la specifica stessa può essere trasformata per ottenere descrizioni equivalenti ma piú vicine all'implementazione. Usando opportuni linguaggi (linguaggi di *programmazione logica*, come, per esempio, il Prolog [11]), una specifica logica può essere espressa in forma eseguibile, e quindi fornire un prototipo del sistema specificato.

La logica è ovviamente importante per l'ingegneria del software anche perché, come accennato prima, è alla base di tutti i metodi formali usati nell'informatica. Inoltre la logica, come linguaggio di specifica, si può integrare con altri linguaggi; per esempio, si possono usare delle espressioni logiche come annotazioni formali per chiarire aspetti del sistema lasciati indeterminati da descrizioni informali.

Esistono diversi tipi di logica, come la logica proposizionale e la logica del primo ordine che vedremo fra poco, ognuno dei quali quali si presta a determinati scopi e campi di applicazione. Ciascuno di questi tipi di logica permette di definire dei *sistemi formali* (o *teorie* formali), ognuno dei quali si basa su un *linguaggio* per mezzo del quale si possono scrivere formule che rappresentano le affermazioni che ci interessano. Un linguaggio viene

descritto dalla propria *sintassi*. Il significato che attribuiamo alle formule è dato dalla *semantica* del linguaggio. La semantica associa i simboli del linguaggio alle entità di cui vogliamo parlare; queste entità costituiscono il *dominio* o *universo di discorso* del linguaggio.

Dato un linguaggio e la sua semantica, un insieme di *regole di inferenza* permette di stabilire se una formula può essere derivata da altre formule, ovvero se una formula è un *teorema* di un certo sistema formale, ovvero se esiste una *dimostrazione* di tale formula.

Le regole di inferenza si riferiscono alla sintassi del linguaggio: possiamo applicare una regola di inferenza ad un insieme di formule senza analizzare il loro significato, e in particolare senza sapere se sono vere o false. Un sistema formale è *corretto* (*sound*) se tutte le formule ottenibili dalle regole d'inferenza sono vere rispetto alla semantica del linguaggio, e *completo* se tutte le formule vere rispetto alla semantica sono ottenibili per mezzo delle regole d'inferenza. Naturalmente la correttezza è un requisito indispensabile per un sistema formale.

Un sistema formale è *decidibile* se esiste un algoritmo che può decidere in un numero finito di passi se una formula è vera (la logica del primo ordine non è decidibile).

Un sistema formale è quindi un apparato di definizioni e regole che ci permette di ragionare formalmente su un qualche settore della conoscenza. Precisiamo che spesso un particolare sistema formale viene indicato come una *logica*. Questa sovrapposizione di termini non crea problemi, perché dal contesto si capisce se si parla *della logica* in generale (la scienza del ragionamento formale) o *di una logica* particolare (un determinato sistema formale).

Le definizioni che verranno date nel séguito sono tratte principalmente da [30], con varie semplificazioni e probabilmente qualche imprecisione.

4.5.1 Calcolo proposizionale

Il calcolo proposizionale è la logica piú semplice. Gli elementi fondamentali del suo linguaggio[4] (non ulteriormente scomponibili) sono le *proposizioni*, cioè delle affermazioni che possono essere vere o false. Il fatto che le proposizioni non siano scomponibili, cioè siano *atomiche*, significa che in una frase come "*il tempo è bello*", il linguaggio del calcolo proposizionale non ci permette di individuare il soggetto e il predicato, poiché questo linguaggio non contiene dei simboli che possano nominare oggetti, proprietà o azioni: in un linguaggio proposizionale possiamo nominare soltanto delle frasi intere. Questo comporta anche che non è possibile mettere in evidenza la struttura comune di certi insiemi di frasi, come, per esempio, "*Aldo è bravo*", "*Beppe è bravo*", e "*Carlo è bravo*". Pertanto, qualsiasi proposizione può essere rappresentata semplicemente da una lettera dell'alfabeto[5], come T per "*il tempo è bello*, A per "*Aldo è bravo*", eccetera.

[4] Piú precisamente, dei linguaggi di tipo proposizionale. Per semplicità, diremo genericamente "*il linguaggio* proposizionale".

[5] O da qualsiasi simbolo o sequenza di caratteri usabile come nome, p.es. '*valvola-chiusa*' nella sez. 4.3.1.

Le proposizioni vengono combinate per mezzo di alcuni operatori per formare frasi piú complesse: gli operatori (chiamati *connettivi*) sono simili alle congiunzioni del linguaggio naturale, da cui hanno ereditato i nomi. Ai connettivi sono associate delle regole (*tabelle* o *funzioni di verità*) che permettono di stabilire la verità delle frasi complesse a partire dalle proposizioni.

Infine, un calcolo proposizionale ha delle *regole d'inferenza* che permettono di derivare alcune frasi a partire da altre frasi. Le regole d'inferenza sono scelte in modo che le frasi derivate per mezzo di esse siano vere, se sono vere le frasi di partenza. Si fa in modo, cioè, che i sistemi formali di tipo proposizionale siano corretti.

Sintassi

Nel calcolo proposizionale un linguaggio è formato da:

- un insieme numerabile $\mathcal{P}$ di *simboli proposizionali* (A,B,C, ...);
- un insieme finito di *connettivi*; per esempio: $\neg$ (*negazione*), $\wedge$ (*congiunzione*), $\vee$ (*disgiunzione*), $\Rightarrow$ (*implicazione*[6]), $\Leftrightarrow$ (*equivalenza* o *coimplicazione*);
- un insieme di *simboli ausiliari* (parentesi e simili);
- un insieme $\mathcal{W}$ di *formule* (dette anche *formule ben formate*, *well-formed formulas*, o *wff*), definito dalle seguenti regole:
  1. un simbolo proposizionale è una formula;
  2. se $\mathcal{A}$ e $\mathcal{B}$ sono formule, allora sono formule anche $(\neg\mathcal{A})$, $(\mathcal{A} \wedge \mathcal{B})$, $(\mathcal{A} \vee \mathcal{B})$, ...
  3. solo le espressioni costruite secondo le due regole precedenti sono formule.

Alle regole sintattiche si aggiungono di solito delle regole ulteriori che permettono di semplificare la scrittura delle formule indicando le priorità dei connettivi, in modo simile a ciò che avviene nella notazione matematica. I connettivi vengono applicati in quest'ordine:

$$\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$$

per cui, ad esempio, la formula

$$A \Leftrightarrow \neg B \vee C \Rightarrow A$$

equivale a

$$A \Leftrightarrow (((\neg B) \vee C) \Rightarrow A)$$

Vediamo quindi che una formula del calcolo proposizionale, analogamente a un'espressione aritmetica, ha una struttura gerarchica: si parte da simboli elementari (i simboli proposizionali) che vengono combinati con operatori unari o binari ottenendo formule che si possono a loro volta combinare, e cosí via.

[6] L'operazione logica associata a questo connettivo si chiama anche *implicazione materiale*, per distinguerla dal concetto di *implicazione logica* che verrà introdotto piú oltre. Ricordiamo anche che a volte viene usato il simbolo $\rightarrow$ (oppure $\supset$) per l'implicazione materiale e il simbolo $\Rightarrow$ per l'implicazione logica.

Semantica

La semantica di un calcolo proposizionale stabilisce le regole che associano un *valore di verità* a ciascuna formula. Il calcolo di questo valore avviene con un metodo di *ricorsione strutturale*: una formula complessa viene scomposta nelle sue formule componenti, fino ai simboli proposizionali; si assegnano i rispettivi valori di verità a ciascuno di essi (per mezzo della *funzione di valutazione*, vedi oltre), sulla base di questi valori si calcolano i valori delle formule di cui fanno parte i rispettivi simboli (per mezzo delle *funzioni di verità*, vedi oltre), e cosí via fino ad ottenere il valore della formula complessiva.

La semantica è quindi data da:

- l'insieme *booleano* $\mathbb{B} = \{\mathbf{T}, \mathbf{F}\}$, contenente i valori *vero* ($\mathbf{T}$, *true*) e *falso* ($\mathbf{F}$, *false*).
- una *funzione di valutazione* $v : \mathcal{P} \to \mathbb{B}$;
- le *funzioni di verità* di ciascun connettivo

$$\begin{array}{rcl} H_\neg & : & \mathbb{B} \to \mathbb{B} \\ H_\wedge & : & \mathbb{B}^2 \to \mathbb{B} \\ & \ldots & \end{array}$$

- una *funzione di interpretazione* $S_v : \mathcal{W} \to \mathbb{B}$ cosí definita:

$$\begin{array}{rcl} S_v(A) & = & v(A) \\ S_v(\neg\mathcal{A}) & = & H_\neg(S_v(\mathcal{A})) \\ S_v(\mathcal{A} \wedge \mathcal{B}) & = & H_\wedge(S_v(\mathcal{A}), S_v(\mathcal{B})) \\ & \ldots & \end{array}$$

  dove $A \in \mathcal{P}$, $\mathcal{A} \in \mathcal{W}$, $\mathcal{B} \in \mathcal{W}$.

La funzione di valutazione assegna un valore di verità a ciascun simbolo proposizionale. Osserviamo che questa funzione è arbitraria, nel senso che viene scelta da chi si vuole servire di un linguaggio logico per rappresentare un certo dominio, in modo da riflettere ciò che si considera vero in tale dominio.

Per esempio, consideriamo le proposizioni A (*Aldo è bravo*), S (*Aldo passa l'esame di Ingegneria del Software*) e T (*il tempo è bello*). A ciascuna proposizione si possono assegnare valori di verità scelti con i criteri ritenuti piú adatti a rappresentare la situazione, per esempio la valutazione di A può essere fatta in base a un giudizio soggettivo sulle capacità di Aldo, la valutazione di S e T può essere fatta in base all'osservazione sperimentale o anche assegnata arbitrariamente, considerando una situazione ipotetica.

Le funzioni di verità dànno il valore di verità restituito dall'operatore logico rappresentato da ciascun connettivo, in funzione dei valori di verità delle formule a cui viene applicato l'operatore. Di solito queste funzioni vengono espresse per mezzo di tabelle di verità come le seguenti, dove $H_\neg$ è la funzione di verità del connettivo *negazione*, etc.:

x	y	$H_\neg(x)$	$H_\wedge(x,y)$	$H_\vee(x,y)$	$H_\Rightarrow(x,y)$	$H_\Leftrightarrow(x,y)$
F	**F**	**T**	**F**	**F**	**T**	**T**
F	**T**	**T**	**F**	**T**	**T**	**F**
T	**F**	**F**	**F**	**T**	**F**	**F**
T	**T**	**F**	**T**	**T**	**T**	**T**

Le funzioni di verità, a differenza della funzione di valutazione, sono parte integrante del linguaggio: poiché definiscono la semantica dei connettivi adottati dal linguaggio, non si possono modificare.

Un insieme di connettivi si dice *completo* se è sufficiente ad esprimere tutte le funzioni di verità possibili. Alcuni insiemi completi sono $\{\neg, \wedge\}$, $\{\neg, \vee\}$ e $\{\neg, \Rightarrow\}$.

La funzione d'interpretazione, infine, calcola il valore di verità di qualsiasi formula, in base alla funzione di valutazione, alle funzioni di verità, ed alla struttura della formula stessa.

Soddisfacibilità e validità

Una formula, in generale, può essere vera o falsa a seconda della funzione di valutazione, cioè del modo in cui vengono assegnati valori di verità ai simboli proposizionali. Esistono però delle formule il cui valore di verità *non dipende* dalla funzione di valutazione, cioè sono vere (o false) per qualsiasi scelta dei valori di verità dei simboli proposizionali. Ovviamente queste formule sono particolarmente utili perché sono di applicabilità piú generale. Nei paragrafi seguenti esprimeremo piú precisamente la dipendenza del valore di verità di una formula dalla funzione di valutazione.

Se, per una formula $\mathcal{F}$ ed una valutazione v, si ha che $S_v(\mathcal{F}) = \mathbf{T}$, si dice che v *soddisfa* $\mathcal{F}$, e si scrive $v \models \mathcal{F}$.

Il simbolo $\models$ non appartiene al linguaggio del calcolo proposizionale, poiché non serve a costruire delle formule, ma è solo un'abbreviazione della frase "soddisfa", che appartiene al metalinguaggio, cioè al linguaggio di cui ci serviamo per parlare del calcolo proposizionale.

Una formula si dice *soddisfacibile* o *consistente* se esiste almeno una valutazione che la soddisfa. Per esempio:

$$A \Rightarrow \neg A \quad \text{per } v(A) = \mathbf{F}$$

Una formula si dice *insoddisfacibile* o *inconsistente* se non esiste alcuna valutazione che la soddisfi. Si dice anche che la formula è una *contraddizione*. Per esempio:

$$A \wedge \neg A$$

Una formula soddisfatta da tutte le valutazioni è una *tautologia*, ovvero è *valida*. La verità di una tautologia non dipende quindi dalla verità delle singole proposizioni che vi appaiono, ma unicamente dalla struttura della formula. Esempi di tautologie sono:

$$A \vee \neg A$$
$$A \Rightarrow A$$
$$\neg\neg A \Rightarrow A$$
$$\neg(A \wedge \neg A)$$
$$(A \wedge B) \Rightarrow A$$
$$A \Rightarrow (A \vee B)$$

Se $\mathcal{A} \Rightarrow \mathcal{B}$ (ove $\mathcal{A}$ e $\mathcal{B}$ sono formule) è una tautologia, si dice che $\mathcal{A}$ *implica logicamente* $\mathcal{B}$, ovvero che $\mathcal{B}$ è *conseguenza logica* di $\mathcal{A}$. Analogamente, se $\mathcal{A} \Leftrightarrow \mathcal{B}$ è una tautologia si dice che $\mathcal{A}$ e $\mathcal{B}$ sono *logicamente equivalenti*.

4.5.2 Teorie formali

Nel calcolo proposizionale è sempre possibile accertarsi se una data formula è valida oppure no: basta calcolare il suo valore di verità per tutte le valutazioni possibili, che per ciascuna formula costituiscono un insieme finito, essendo finiti gli insiemi dei simboli proposizionali nella formula, delle loro occorrenze, e dei loro possibili valori di verità. Questo metodo diretto di verifica della validità fa riferimento alla semantica del linguaggio usato.

Se il numero di simboli proposizionali in una formula è grande, la verifica diretta può essere impraticabile. Inoltre la verifica diretta è generalmente impossibile nelle logiche piú potenti del calcolo proposizionale (come la logica del primo ordine, che vedremo fra poco), la cui semantica può comprendere insiemi infiniti di valori. Si pone quindi il problema di *dedurre* la verità di una formula non attraverso il calcolo diretto del suo valore di verità, ma attraverso certe relazioni, basate sulla sua sintassi, con altre formule. Informalmente, il meccanismo di deduzione si può descrivere in questo modo:

1. si sceglie un insieme di formule che consideriamo valide *a priori*, senza necessità di dimostrazione, oppure che verifichiamo direttamente per mezzo della semantica;
2. si definiscono delle regole (dette *d'inferenza*) che, date alcune formule valide aventi una certa struttura, permettono di scrivere nuove formule valide;
3. a partire dalle formule introdotte al punto 1, si costruiscono delle catene di formule applicando ripetutamente le regole d'inferenza;
4. ogni formula appartenente a una delle catene cosí costruite si considera dimostrata sulla base delle formule che la precedono nella catena.

Regole di inferenza

Una *regola di inferenza* è una relazione fra formule; per ogni n-upla di formule che soddisfa una regola di inferenza R, una determinata formula della n-upla viene chiamata *conseguenza diretta* delle altre formule della n-upla *per effetto di* R. Per esempio, una regola di inferenza potrebbe essere l'insieme delle triple aventi la forma $\langle \mathcal{B}, \mathcal{A}, \mathcal{A} \Rightarrow \mathcal{B} \rangle$. Questa regola si scrive generalmente in questa forma:

$$\frac{\mathcal{A} \qquad \mathcal{A} \Rightarrow \mathcal{B}}{\mathcal{B}}$$

che si legge "$\mathcal{B}$ *è conseguenza diretta di* $\mathcal{A}$ *e di* $\mathcal{A} \Rightarrow \mathcal{B}$". Meno sinteticamente: "Se $\mathcal{A}$ è valida ed $\mathcal{A}$ implica $\mathcal{B}$, allora possiamo dedurre che $\mathcal{B}$ è valida".

Dimostrazioni e teoremi

Possiamo ora introdurre il concetto di *teoria* (o *sistema*) *formale*, nell'àmbito del quale si definiscono i concetti di *dimostrazione* e di *teorema*.

Una teoria formale è data da

1. un linguaggio $\mathcal{L}$;
2. un insieme $\mathbf{A}$ (eventualmente infinito) di formule di $\mathcal{L}$ chiamate *assiomi*;
3. un insieme finito di regole di inferenza fra formule di $\mathcal{L}$.

Se Γ è un insieme di formule, dette *ipotesi* o *premesse*, $\mathcal{F}$ è una formula da dimostrare, e Δ è una sequenza di formule, allora si dice che Δ è una *dimostrazione* (o *deduzione*) *di* $\mathcal{F}$ *da* Γ se l'ultima formula di Δ è $\mathcal{F}$ e ciascun'altra: o (i) è un assioma, o (ii) è una premessa, o (iii) è conseguenza diretta di formule che la precedono nella sequenza Δ.

Si dice quindi che $\mathcal{F}$ *segue da* Γ, o è *conseguenza* di Γ, e si scrive

$$\Gamma \vdash \mathcal{F}$$

Se l'insieme Γ delle premesse è vuoto, allora la sequenza Δ è una **dimostrazione di** $\mathcal{F}$, ed $\mathcal{F}$ è un *teorema*, e si scrive

$$\vdash \mathcal{F}$$

Quindi, in una teoria formale, una dimostrazione è una sequenza di formule tali che ciascuna di esse o è un assioma o è conseguenza diretta di alcune formule precedenti, e un teorema è una formula che si può dimostrare ricorrendo solo agli assiomi, senza ipotesi aggiuntive.

Osservazione. Notare la differenza fra "*dimostrazione di* $\mathcal{F}$" e "*dimostrazione di* $\mathcal{F}$ *da* Γ".

Una teoria formale per il calcolo proposizionale

Una semplice teoria formale per il calcolo proposizionale può essere definita come segue [30]:

- Il linguaggio $\mathcal{L}$ è costituito dalle formule ottenute a partire dai simboli proposizionali, dai connettivi $\neg$ e $\Rightarrow$, e dalle parentesi.
- Gli assiomi sono tutte le espressioni date dai seguenti *schemi di assiomi*, dove $\mathcal{A}, \mathcal{B}, \mathcal{C} \in \mathcal{W}$:

$$\mathcal{A} \Rightarrow (\mathcal{B} \Rightarrow \mathcal{A}) \tag{4.1}$$

$$(\mathcal{A} \Rightarrow (\mathcal{B} \Rightarrow \mathcal{C})) \Rightarrow ((\mathcal{A} \Rightarrow \mathcal{B}) \Rightarrow (\mathcal{A} \Rightarrow \mathcal{C})) \tag{4.2}$$

$$(\neg\mathcal{B} \Rightarrow \neg\mathcal{A}) \Rightarrow ((\neg\mathcal{B} \Rightarrow \mathcal{A}) \Rightarrow \mathcal{B}) \tag{4.3}$$

Questa teoria formale ha quindi un insieme infinito di assiomi; osserviamo anche che qualsiasi assioma ottenuto da questi schemi è una tautologia.

- L'unica regola d'inferenza è il *modus ponens* (MP): una formula $\mathcal{B}$ è conseguenza diretta di $\mathcal{A}$ e $\mathcal{A} \Rightarrow \mathcal{B}$. Si scrive anche

$$\frac{\mathcal{A} \qquad \mathcal{A} \Rightarrow \mathcal{B}}{\mathcal{B}}$$

Un'utile proprietà di questa teoria formale è il *Teorema della deduzione*: *se* Γ *è un insieme di wff,* $\mathcal{A}$ *e* $\mathcal{B}$ *sono wff, e* $\Gamma \cup \{\mathcal{A}\} \vdash \mathcal{B}$*, allora* $\Gamma \vdash \mathcal{A} \Rightarrow \mathcal{B}$. In particolare, se $\mathcal{A} \vdash \mathcal{B}$, allora $\vdash \mathcal{A} \Rightarrow \mathcal{B}$. Cioè, si può affermare che $\mathcal{A}$ implica logicamente $\mathcal{B}$ se $\mathcal{B}$ è dimostrabile da $\mathcal{A}$, coerentemente col comune modo di dimostrare i teoremi in matematica.

Osservazione. Il Teorema della deduzione è, propriamente, un metateorema, perché afferma una proprietà di un sistema formale, mentre un teorema è una formula dimostrabile nell'àmbito del sistema stesso.

Infine, si può dimostrare che questa teoria formale è corretta e completa, cioè che ogni teorema di questa teoria è una tautologia, e viceversa.

4.5.3 Logica del primo ordine

La *logica del primo ordine* (anche *FOL*, *first order logic*), o *logica dei predicati*[7] permette di formare delle frasi (*formule*) in cui è possibile riferirsi a entità individuali (*oggetti* o *individui*), sia specificamente, per mezzo di simboli (*costanti* e *funzioni*) che denotano particolari entità, sia genericamente, per mezzo di simboli (*variabili*) che si riferiscono a individui non specificati (analogamente ai pronomi indefiniti nel linguaggio naturale). Le formule del linguaggio, inoltre, possono essere *quantificate* rispetto alle variabili che vi compaiono, cioè si può indicare se determinate proprietà valgono per tutti i valori di tali variabili o solo per alcuni.

Le frasi piú semplici che possiamo esprimere nel linguaggio della logica del primo ordine sono del tipo "gli oggetti a, b, c, ... sono legati dalla relazione p" (o, come caso particolare, "l'oggetto a gode della proprietà p"). Queste formule vengono combinate per costruire frasi piú complesse, usando i connettivi ed i quantificatori.

Per esempio, consideriamo queste frasi:

1. "*Aldo è bravo*", "*Beppe è bravo*", "*Carlo è bravo*".
2. "*Aldo passa Ingegneria del Software*", "*Beppe passa Sistemi Operativi e Reti*", "*Carlo passa Epistemologia Generale*".

Le frasi del gruppo 1 dicono che certi individui godono di una certa proprietà, ovvero che Aldo, Beppe e Carlo appartengono all'insieme degli studenti bravi. Le frasi del gruppo

[7] Il termine *logica dei predicati* è piú generale di *logica del primo ordine*, ma in questo testo i due termini sono usati come sinonimi.

2 dicono che esiste una certa relazione fra certi individui e certi altri individui (non si fa distinzione fra entità animate e inanimate), ovvero che le coppie (Aldo, Ingegneria del Software), (Beppe, Sistemi Operativi e Reti), e (Carlo, Epistemologia Generale) appartengono all'insieme di coppie ordinate tali che l'individuo nominato dal primo elemento della coppia abbia superato l'esame nominato dal secondo elemento.

Sia le frasi del primo gruppo che quelle del secondo affermano che certi *predicati*, cioè proprietà o relazioni, valgono per certe entità individuali, e si possono scrivere sinteticamente come:

$$b(A),\ b(B),\ b(C),\ p(A, I),\ p(B, S),\ p(C, E)$$

dove il *simbolo di predicato* b sta per "è bravo", p per "passa l'esame di", il *simbolo di costante* A per "Aldo", e cosí via.

Queste frasi sono istanze particolari delle formule $b(x)$ e $p(x, y)$, dove le variabili x e y sono dei simboli segnaposto che devono essere sostituiti da nomi di individui per ottenere delle frasi cui si possa assegnare un valore di verità. È bene sottolineare che la formula $b(x)$ non significa "*qualcuno è bravo*": la formula non significa niente, non è né vera né falsa, finché il simbolo x non viene sostituito da un'espressione che denoti un particolare individuo, per esempio "Aldo" o anche "il padre di Aldo". Dopo questa sostituzione, la formula può essere vera o falsa, ma ha comunque un valore definito.

Se vogliamo esprimere nella logica del primo ordine la frase "*qualcuno è bravo*", dobbiamo usare un nuovo tipo di simbolo, il quantificatore esistenziale: $\exists x\, b(x)$, cioè "esiste qualcuno che è bravo". Questa formula ha significato cosí com'è, perché afferma una proprietà dell'insieme degli studenti bravi (la proprietà di non essere vuoto).

Una formula come $b(x)$, dove, cioè, appaiono variabili non quantificate, si dice *aperta*, mentre una formula come $\exists x\, b(x)$, con le variabili quantificate, si dice *chiusa*.

Sintassi

Quanto sopra viene espresso piú formalmente dicendo che un linguaggio del primo ordine è costituito da:

- un insieme numerabile $\mathcal{V}$ di *variabili* (x, y, z, ...);
- un insieme numerabile $\mathcal{F}$ di *simboli di funzione* (f, g, h, ...); a ciascun simbolo di funzione è associato il numero di argomenti (*arietà*) della funzione: $\mathcal{F} = \mathcal{F}_0 \cup \mathcal{F}_1 \cup \mathcal{F}_2 \cup \ldots$, dove $\mathcal{F}_n$ è l'insieme dei simboli n-ari di funzione; le funzioni di arietà zero sono chiamate *costanti* (a, b, c, ...);
- un insieme $\mathcal{T}$ di *termini* (t_1, t_2, t_3, ...), definito dalle seguenti regole:
  1. una variabile è un termine;
  2. se f è un simbolo n-ario di funzione e $t_1, \ldots, t_n$ sono termini, allora $f(t_1, \ldots, t_n)$ è un termine; in particolare, un simbolo c di arietà nulla è un termine (*costante*);
  3. solo le espressioni costruite secondo le due regole precedenti sono termini.

- un insieme numerabile $\mathcal{P}$ di *simboli di predicato* (p, q, r, ...); a ciascun simbolo di predicato è associato il numero di argomenti (*arietà*) del predicato: $\mathcal{P} = \mathcal{P}_0 \cup \mathcal{P}_1 \cup \mathcal{P}_2 \cup \ldots$, dove $\mathcal{P}_n$ è l'insieme dei simboli n-ari di predicato; i predicati di arietà zero corrispondono ai simboli proposizionali del calcolo proposizionale;
- un insieme finito di *connettivi*; per esempio:

$$\neg, \wedge, \vee, \Rightarrow, \ldots$$

- un insieme finito di *quantificatori*; per esempio:

$$\forall, \exists$$

- un insieme di *simboli ausiliari* (parentesi e simili);
- un insieme $\mathcal{W}$ di *formule*, definito dalle seguenti regole:
  1. se p è un simbolo n-ario di predicato e $t_1, \ldots, t_n$ sono termini, allora $p(t_1, \ldots, t_n)$ è una formula (detta *atomica*, in questo caso);
  2. se $\mathcal{A}$ e $\mathcal{B}$ sono formule, allora sono formule anche $(\neg\mathcal{A})$, $(\mathcal{A} \wedge \mathcal{B})$, $(\mathcal{A} \vee \mathcal{B})$, ...
  3. se $\mathcal{A}$ è una formula e x è una variabile, allora sono formule anche $(\forall x\mathcal{A})$ e $(\exists x\mathcal{A})$. Le formule $(\forall x\mathcal{A})$ e $(\exists x\mathcal{A})$ sono dette *formule quantificate* ed $\mathcal{A}$ è il *campo* del rispettivo quantificatore.
  4. solo le espressioni costruite secondo le regole precedenti sono formule.

Le priorità nell'ordine di applicazione dei connettivi è la stessa vista per il calcolo proposizionale, salvo che la priorità dei quantificatori è intermedia fra quella di $\vee$ e quella di $\Rightarrow$. Ricordiamo però che questa convenzione non è universalmente applicata.

Semantica

La semantica di una logica del primo ordine è data da:

- l'insieme $\mathbb{B} = \{\mathbf{T}, \mathbf{F}\}$;
- le *funzioni di verità* di ciascun connettivo;
- un'*interpretazione* $I = (\mathcal{D}, \Phi, \Pi)$ data da
  - l'insieme non vuoto $\mathcal{D}$, detto *dominio* dell'interpretazione;
  - la *funzione di interpretazione delle funzioni* $\Phi : \mathcal{F} \to \mathcal{F}_\mathcal{D}$, dove $\mathcal{F}_\mathcal{D}$ è l'insieme delle funzioni su $\mathcal{D}$. Φ assegna a ciascun simbolo n-ario di funzione una funzione $\mathcal{D}^n \to \mathcal{D}$;
  - la *funzione di interpretazione dei predicati* $\Pi : \mathcal{P} \to \mathcal{R}_\mathcal{D}$, dove $\mathcal{R}_\mathcal{D}$ è l'insieme delle relazioni su $\mathcal{D}$; Π assegna a ciascun simbolo n-ario di predicato una funzione $\mathcal{D}^n \to \mathbb{B}$;
- un *assegnamento di variabili* $\xi : \mathcal{V} \to \mathcal{D}$;

Tabella 4.2: Elementi della semantica.

simbolo	definizione
$\mathcal{D}$	dominio
$\Phi : \mathcal{F} \to \mathcal{F}_\mathcal{D}$	interpretazione delle funzioni
$\Pi : \mathcal{P} \to \mathcal{R}_\mathcal{D}$	interpretazione dei predicati
$\xi : \mathcal{V} \to \mathcal{D}$	assegnamento di variabili
$\Xi : \mathcal{T} \to \mathcal{D}$	assegnamento di termini
$S_{I,\xi} : \mathcal{W} \to \mathbb{B}$	interpretazione delle formule

- un *assegnamento di termini* $\Xi : \mathcal{T} \to \mathcal{D}$ cosí definito:

$$\begin{aligned} \Xi(x) &= \xi(x) \\ \Xi(f(t_1, \ldots, t_n)) &= \Phi(f)(\ \Xi(t_1), \ldots, \Xi(t_n)\ ) \end{aligned}$$

  dove $x \in \mathcal{V}$, $t_i \in \mathcal{T}$, $f \in \mathcal{F}$;
- una *funzione di interpretazione* $S_{I,\xi} : \mathcal{W} \to \mathbb{B}$ cosí definita:

$$\begin{aligned} S_{I,\xi}(p(t_1, \ldots, t_n)) &= \Pi(p)(\ \Xi(t_1), \ldots, \Xi(t_n)\ ) \\ S_{I,\xi}(\neg\mathcal{A}) &= H_\neg(S_{I,\xi}(\mathcal{A})) \\ S_{I,\xi}(\mathcal{A} \wedge \mathcal{B}) &= H_\wedge(S_{I,\xi}(\mathcal{A}), S_{I,\xi}(\mathcal{B})) \\ &\ldots \\ S_{I,\xi}(\exists x\mathcal{A}) &= \mathbf{T} \\ &\quad \text{se e solo se esiste un } d \in \mathcal{D} \text{ tale che} \\ &\quad [S_{I,\xi}]_{x/d}(\mathcal{A}) = \mathbf{T} \\ S_{I,\xi}(\forall x\mathcal{A}) &= \mathbf{T} \\ &\quad \text{se e solo se per ogni } d \in \mathcal{D} \text{ si ha} \\ &\quad [S_{I,\xi}]_{x/d}(\mathcal{A}) = \mathbf{T} \end{aligned}$$

  dove $p \in \mathcal{P}$, $t_i \in \mathcal{T}$, $\mathcal{A}, \mathcal{B} \in \mathcal{W}$, $x \in \mathcal{V}$, e $[S_{I,\xi}]_{x/d}$ è la funzione di interpretazione uguale a $S_{I,\xi}$ eccetto che assegna alla variabile x il valore d.

Gli elementi fondamentali della semantica sono riassunti nella tab. 4.2.

Il dominio è l'insieme degli oggetti di cui vogliamo parlare. Per esempio, se volessimo parlare dell'aritmetica, il dominio sarebbe l'insieme dei numeri interi.

La funzione d'interpretazione delle funzioni stabilisce il significato dei simboli che usiamo nel nostro linguaggio per denotare le funzioni. Nell'esempio dell'aritmetica, una funzione di interpretazione delle funzioni potrebbe associare la funzione aritmetica *somma* ($somma \in \mathcal{F}_\mathcal{D}$) al simbolo di funzione f ($f \in \mathcal{F}$), cioè $\Phi(f) = somma$. In particolare, Φ stabilisce il significato delle costanti, per esempio possiamo associare al simbolo a il numero *zero*, al simbolo b il numero *uno*, e cosí via. Di solito, quando si usa la logica per parlare di un argomento ove esiste una notazione tradizionale, si cerca di usare quella notazione, a meno che non si voglia sottolineare la distinzione fra linguaggio e dominio (come stiamo facendo qui). Quindi in genere è possibile usare il simbolo '+' per la somma, il simbolo '1' per il numero *uno*, eccetera.

Tabella 4.3: Esempio di interpretazione.

simbolo	funzione	interpretazione	simbolo comune
f	Φ	somma	$+$
b	Φ	uno	1
x	ξ	tre	3
p	Π	minore o uguale	$\leq$

La funzione d'interpretazione dei predicati stabilisce il significato dei simboli che denotano le relazioni. Per esempio, una funzione di interpretazione dei predicati può associare la relazione *minore o uguale* al simbolo di predicato p.

L'assegnamento di variabili stabilisce il significato delle variabili. Per esempio, possiamo assegnare il valore *tre* alla variabile x.

L'assegnamento di termini stabilisce il significato dei termini. Per esempio, dato il termine $f(b, x)$, e supponendo che $\Phi(f) = \mathit{somma}$, $\Phi(b) = \mathit{uno}$, $\xi(x) = \mathit{tre}$, allora $\Xi(f(b, x)) = \mathit{quattro}$.

Infine, la funzione d'interpretazione (delle formule) stabilisce il significato delle formule di qualsivoglia complessità. Come nel calcolo proposizionale, la funzione di interpretazione ha una definizione ricorsiva e si applica analizzando ciascuna formula nelle sue componenti. Le formule piú semplici (formule atomiche) sono costituite da simboli di predicato applicati ad n-uple di termini, quindi il valore di verità di una formula atomica (fornito dalla funzione Π applicata agli argomenti del predicato) dipende dall'assegnamento dei termini.

La definizione della funzione di interpretazione per le formule quantificate rispecchia la comune nozione di quantificazione esistenziale ed universale, a dispetto della notazione un po' oscura qui adottata. L'espressione $[S_{I,\xi}]_{x/d}$ è definita come "la funzione di interpretazione uguale a $S_{I,\xi}$ eccetto che assegna alla variabile x il valore d". Questo significa che, per vedere se una formula quantificata sulla variabile x è vera, non ci interessa *il* valore attribuito a x dal particolare assegnamento ξ, ma *l'insieme* dei possibili valori di x, indipendentemente dall'assegnamento: per la quantificazione esistenziale vediamo se almeno uno dei valori possibili soddisfa la formula compresa nel campo del quantificatore, per la quantificazione universale vediamo se tutti i valori possibili la soddisfano. In ambedue i casi, per le altre variabili si considerano i valori assegnati da ξ.

Come esempio di interpretazione, data la formula $p(f(b, x), x) \wedge p(b, x)$, si ha $S_{I,\xi}(p(f(b, x), x) \wedge p(b, x)) = \mathbf{F}$, se l'interpretazione I e l'assegnamento di variabili ξ sono compatibili con le interpretazioni ed assegnamenti visti sopra. Si verifica facilmente che I e ξ trasformano la formula logica considerata nell'espressione $1 + 3 \leq 3 \wedge 1 \leq 3$. La Tab. 4.3 riassume le interpretazioni dei vari simboli.

Soddisfacibilità e validità

Nel calcolo proposizionale l'interpretazione di una formula dipende solo dalla sua struttura e, in generale, dalla funzione di valutazione. Nella logica dei predicati le formule sono piú

complesse e il loro valore di verità dipende, oltre che dalla struttura della formula, anche dal dominio di interpretazione, dalle funzioni d'interpretazione dei simboli di funzione e di predicato, e dalla funzione di assegnamento di variabili. Il dominio e le funzioni di interpretazione costituiscono, come abbiamo visto, l'interpretazione del linguaggio e ne definiscono l'aspetto strutturale, invariabile. Anche nella logica dei predicati ci interessa studiare la dipendenza del valore di verità delle formule dall'interpretazione del linguaggio e dall'assegnamento di variabili. Di séguito le definizioni relative.

Una formula $\mathcal{A}$ è *soddisfacibile in un'interpretazione* I se e solo se esiste un assegnamento di variabili ξ tale che $S_{I,\xi}(\mathcal{A}) = \mathbf{T}$. Si dice allora che l'interpretazione I *soddisfa* $\mathcal{A}$ con assegnamento di variabili ξ, e si scrive $I \overset{\xi}{\models} \mathcal{A}$.

Una formula $\mathcal{A}$ è *soddisfacibile* (tout-court) se e solo se esiste un'interpretazione I in cui $\mathcal{A}$ è soddisfacibile.

Una formula $\mathcal{A}$ è *valida in un'interpretazione* (o *vera in un'interpretazione*) I se e solo se $S_{I,\xi}(\mathcal{A}) = \mathbf{T}$ per ogni assegnamento di variabili ξ. Si dice quindi che I è un *modello* di $\mathcal{A}$, e si scrive $I \models \mathcal{A}$.

Osserviamo che una formula *aperta* $\mathcal{A}$, cioè contenente variabili non quantificate, è valida in un'interpretazione I se e soltanto se è valida la sua *chiusura universale*, cioè la formula ottenuta da $\mathcal{A}$ quantificando universalmente le sue variabili libere.

Una formula $\mathcal{A}$ è (*logicamente*) *valida* se e solo se è valida per ogni interpretazione I, e si scrive $\models \mathcal{A}$.

Nella logica del primo ordine, si chiamano tautologie le formule che si possono ottenere da una tautologia del calcolo proposizionale sostituendo uniformemente ogni simbolo proposizionale con una formula del primo ordine. Le tautologie sono quindi formule valide, ma non tutte le formule valide sono tautologie.

I concetti di validità e soddisfacibilità si estendono a insiemi di formule: un insieme di formule è valido o soddisfacibile se lo è la loro congiunzione.

Una formula $\mathcal{A}$ *implica logicamente* $\mathcal{B}$, ovvero $\mathcal{B}$ è *conseguenza logica* di $\mathcal{A}$ se e solo se $\mathcal{A} \Rightarrow \mathcal{B}$ è valida. Una formula $\mathcal{A}$ *è logicamente equivalente* a $\mathcal{B}$, se e solo se $\mathcal{A} \Leftrightarrow \mathcal{B}$ è valida.

Una teoria formale per la FOL

Una semplice teoria formale per la FOL può essere definita come segue [30]:

- il linguaggio $\mathcal{L}$ è un linguaggio del primo ordine che usa i connettivi $\neg$ e $\Rightarrow$, il quantificatore $\forall$, e le parentesi[8];

[8] La quantificazione esistenziale si può esprimere usando il quantificatore universale, se poniamo $\exists x \mathcal{A}$ equivalente a $\neg(\forall x(\neg\mathcal{A}))$.

- gli assiomi sono divisi in *assiomi logici* ed *assiomi propri* (o *non logici*); gli assiomi logici sono tutte le espressioni date dai seguenti *schemi*:

$$\mathcal{A} \Rightarrow (\mathcal{B} \Rightarrow \mathcal{A}) \tag{4.4}$$
$$(\mathcal{A} \Rightarrow (\mathcal{B} \Rightarrow \mathcal{C})) \Rightarrow ((\mathcal{A} \Rightarrow \mathcal{B}) \Rightarrow (\mathcal{A} \Rightarrow \mathcal{C})) \tag{4.5}$$
$$(\neg\mathcal{B} \Rightarrow \neg\mathcal{A}) \Rightarrow ((\neg\mathcal{B} \Rightarrow \mathcal{A}) \Rightarrow \mathcal{B}) \tag{4.6}$$
$$\forall x \mathcal{A}(x) \Rightarrow \mathcal{A}(t) \tag{4.7}$$
$$\forall x(\mathcal{A} \Rightarrow \mathcal{B}) \Rightarrow (\mathcal{A} \Rightarrow \forall x \mathcal{B}) \tag{4.8}$$

  Gli assiomi propri sono specifici di ciascuna teoria, per esempio gli assiomi dell'aritmetica di Peano o della teoria dei gruppi.
- le regole d'inferenza sono:
  - *modus ponens* come nel calcolo proposizionale;
  - *generalizzazione*

$$\frac{\mathcal{A}}{\forall x \mathcal{A}} .$$

Negli assiomi, $\mathcal{A}(x)$ rappresenta una formula contenente la variabile x (ed eventualmente altre variabili), ed $\mathcal{A}(t)$ è la formula ottenuta da $\mathcal{A}(x)$ sostituendo x col termine t.

Allo schema di assioma 4.7 bisogna aggiungere la restrizione che t sia *libero per x in $\mathcal{A}$*, cioè che non contenga variabili che in $\mathcal{A}$ abbiano un quantificatore universale il cui campo d'azione includa occorrenze di x; questo vincolo impedisce che variabili libere nel termine t diventino quantificate quando t viene sostituito a x. Supponiamo, per esempio, di voler dimostrare la formula

$$\forall x(\neg\forall y(x = y)) \Rightarrow \neg(\forall y(y = y))$$

nel dominio dei numeri interi.

Questa formula significa che "*se per ogni x esiste un y diverso da x, allora non è vero che ogni y è uguale a se stesso*", ed è falsa, date le note proprietà della relazione di uguaglianza. Però la formula ha la stessa struttura dello schema 4.7, come si verifica sostituendo $\neg\forall y(x = y)$ ad $\mathcal{A}(x)$ e $\neg(\forall y(y = y))$ al posto di $\mathcal{A}(t)$. Ma lo schema non è applicabile, perché richiede che la metavariabile t venga sostituita dal termine y, che non è libero per x nella formula $\neg\forall y(x = y)$; infatti y è quantificata nella sottoformula $\neg\forall y(x = y)$ (che sostituisce $\mathcal{A}(x)$), e il campo d'azione del suo quantificatore comprende una occorrenza di x.

Allo schema di assioma 4.8 bisogna aggiungere la restrizione che $\mathcal{A}$ non contenga occorrenze libere di x. Per esempio, se $\mathcal{A}$ e $\mathcal{B}$ corrispondono tutte e due alla formula $p(x)$, dove $p(x)$ viene interpretato come "*x è pari*", l'applicazione dell'assioma 4.8 porterebbe alla formula $\forall x(p(x) \Rightarrow p(x)) \Rightarrow (p(x) \Rightarrow \forall x\, p(x))$, che è falsa: l'antecedente dell'implicazione principale $\forall x(p(x) \Rightarrow p(x))$ significa che "*per ogni x la parità implica la parità*", e il conseguente $(p(x) \Rightarrow \forall x\, p(x))$ significa che "*la parità di un numero implica la parità di tutti i numeri*".

Un sistema formale può essere privo di assiomi propri, e in questo caso viene chiamato *calcolo dei predicati*.

In ogni calcolo dei predicati, tutti i teoremi sono formule logicamente valide e viceversa (*Teorema di completezza* di Gödel).

Il sistema formale qui esposto ha un numero relativamente grande di schemi di assiomi e poche regole di inferenza: i sistemi di questo tipo vengono spesso detti *alla Hilbert*. Altri sistemi formali, come la *deduzione naturale* o il *calcolo dei sequenti* (v. oltre), hanno pochi assiomi e un maggior numero di regole d'inferenza.

4.5.4 Esempio di specifica e verifica formale

Consideriamo il problema dell'ordinamento di un vettore (esempio tratto da [16]). Vogliamo specificare la relazione fra un vettore arbitrario x di $N > 2$ elementi ed il vettore y ottenuto ordinando x in ordine crescente, supponendo che gli elementi del vettore abbiano valori distinti. Possiamo esprimere questa relazione cosí:

$$\begin{aligned}
\mathit{ord}(x,y) &\Leftrightarrow \mathit{permutazione}(x,y) \wedge \mathit{ordinato}(y) \\
\mathit{permutazione}(x,y) &\Leftrightarrow \forall k((1 \le k \wedge k \le N) \Rightarrow \\
&\quad \exists i(1 \le i \wedge i \le N \wedge y_i = x_k) \wedge \\
&\quad \exists j(1 \le j \wedge j \le N \wedge x_j = y_k)) \\
\mathit{ordinato}(x) &\Leftrightarrow \forall k(1 \le k \wedge k < N \Rightarrow x_k \le x_{k+1})
\end{aligned}$$

Possiamo verificare la correttezza di un programma che ordina un vettore di N elementi rispetto alla specifica. Trascuriamo la parte di specifica relativa alla permutazione di un vettore. L'ordinamento di un vettore `v` può essere ottenuto col seguente frammento di programma, che implementa l'algoritmo bubblesort, dove $M = N - 1$ (ricordiamo che in C++ gli indici di un array di N elementi vanno da 0 a $N - 1$):

```
for (i = 0; i < M; i++) {           // 0 ≤ i < M
    for (j = 0; j < M-i; j++) {     // 0 ≤ j < M − i
        if (v[j] > v[j+1]) {
            t = v[j];
            v[j] = v[j+1];
            v[j+1] = t;
        } // v_j ≤ v_{j+1}                                   (i)
    } // ∀k(M − i − 1 ≤ k < M ⇒ v_k ≤ v_{k+1})              (ii)
} // ∀k(0 ≤ k < M ⇒ v_k ≤ v_{k+1})                          (iii)
```

L'asserzione (i) vale dopo l'esecuzione dell'istruzione `if`, per il valore corrente di `j`. Le asserzioni (ii) e (iii) valgono, rispettivamente, all'uscita del loop interno (quando sono

stati ordinati gli ultimi $i+2$ elementi) e del loop esterno (quando sono stati ordinati tutti gli elementi). Le altre due asserzioni esprimono gli intervalli dei valori assunti da `i` e `j`. La forma $A \leq B < C$ è un'abbreviazione di $A \leq B \wedge B < C$.

Per brevità di esposizione, giustifichiamo informalmente l'asserzione (i) considerando le operazioni svolte nel corpo dell'istruzione `if`. Una verifica formale richiede una definizione della semantica dell'assegnamento, e quindi un modello formale della memoria di un calcolatore.

Possiamo verificare la formula (ii) per induzione sul valore di i, la variabile che controlla il ciclo esterno.

Per $i = 0$ (base dell'induzione), la (ii) diventa

$$\forall k(M - 1 \leq k < M \Rightarrow v_k \leq v_{k+1})$$

La variabile k assume solo il valore $M - 1$, e j varia fra 0 ed $M - 1$, pertanto all'uscita del ciclo si ha dalla (i) che $v_{M-1} \leq v_M$, e la (ii) è verificata per $i = 0$.

Per un $\bar{\imath}$ arbitrario purché minore di $M - 1$, la (ii) (che è vera per l'ipotesi induttiva) assicura che gli ultimi $\bar{\imath} + 2$ elementi del vettore sono ordinati. Sono state eseguite $M - \bar{\imath}$ iterazioni del ciclo interno, e (dalla (i)) $v_{M-\bar{\imath}-1} < v_{M-\bar{\imath}}$. Per $i = \bar{\imath}+1$, il ciclo interno viene eseguito $M - \bar{\imath} - 1$ volte, e all'uscita del ciclo $v_{M-\bar{\imath}-2} < v_{M-\bar{\imath}-1}$. Quindi gli ultimi $i + 2$ elementi sono ordinati. La (ii) è quindi dimostrata.

La formula (iii) si ottiene dalla (ii) ponendovi $i = M-1$, essendo $M-1$ l'ultimo valore assunto da i. Risulta quindi che all'uscita del ciclo esterno il vettore è ordinato, q.e.d.

Osserviamo che sono state sviluppate delle logiche destinate espressamente alla verifica del software, come la logica di Floyd e Hoare [14, 22].

4.5.5 Un'altra teoria formale per la FOL: il calcolo dei sequenti

Il *calcolo dei sequenti* (Gentzen) è un sistema formale le cui regole di inferenza operano su espressioni (i *sequenti*), aventi questa struttura:

$$\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n \vdash \mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_m \ ,$$

ove gli $\mathcal{A}_i$ sono detti *antecedenti* (o, collettivamente, *l'antecedente*) e i $\mathcal{B}_i$ *conseguenti* (*il conseguente*). Il simbolo '$\vdash$', che qui chiameremo *simbolo di sequente*, si può leggere "*comporta*" o "*dimostra*" (*yields*, *entails*). Ciascun $\mathcal{A}_i$ e $\mathcal{B}_i$, a sua volta, è una formula qualsiasi del linguaggio visto nelle sezioni precedenti (o anche di altri linguaggi). Osserviamo che il simbolo di sequente *non appartiene* al linguaggio di cui vogliamo dimostrare teoremi.

Informalmente, un sequente può essere considerato come una notazione alternativa per questa espressione:

$$\mathcal{A}_1 \wedge \mathcal{A}_2 \wedge \ldots \wedge \mathcal{A}_n \Rightarrow \mathcal{B}_1 \vee \mathcal{B}_2 \vee \ldots \vee \mathcal{B}_m \ ,$$

Tabella 4.4: Le regole d'inferenza nel calcolo dei sequenti.

$$\frac{}{\Gamma,\mathcal{A}\vdash\mathcal{A},\Delta}\text{axm} \qquad \frac{\Gamma\vdash\Delta,\mathcal{A}\ \ \mathcal{A},\Gamma\vdash\Delta}{\Gamma\vdash\Delta}\ \text{cut} \qquad \frac{\mathcal{A},\mathcal{A},\Gamma\vdash\Delta}{\mathcal{A},\Gamma\vdash\Delta}\text{ctr L} \qquad \frac{\Gamma\vdash\Delta,\mathcal{A},\mathcal{A}}{\Gamma\vdash\Delta,\mathcal{A}}\text{ctr R}$$

$$\frac{\Gamma\vdash\Delta,\mathcal{A}}{\neg\mathcal{A},\Gamma\vdash\Delta}\neg L \qquad \frac{\mathcal{A},\Gamma\vdash\Delta}{\Gamma\vdash\Delta,\neg\mathcal{A}}\neg R \qquad \frac{\mathcal{A},\mathcal{B},\Gamma\vdash\Delta}{\mathcal{A}\wedge\mathcal{B},\Gamma\vdash\Delta}\wedge L \qquad \frac{\Gamma\vdash\Delta,\mathcal{A}\ \ \Gamma\vdash\Delta,\mathcal{B}}{\Gamma\vdash\Delta,\mathcal{A}\wedge\mathcal{B}}\wedge R$$

$$\frac{\mathcal{A},\Gamma\vdash\Delta\ \ \mathcal{B},\Gamma\vdash\Delta}{\mathcal{A}\vee\mathcal{B},\Gamma\vdash\Delta}\vee L \qquad \frac{\Gamma\vdash\Delta,\mathcal{A},\mathcal{B}}{\Gamma\vdash\Delta,\mathcal{A}\vee\mathcal{B}}\vee R \qquad \frac{\Gamma\vdash\Delta,\mathcal{A}\ \ \mathcal{B},\Gamma\vdash\Delta}{\mathcal{A}\Rightarrow\mathcal{B},\Gamma\vdash\Delta}\Rightarrow L \qquad \frac{\mathcal{A},\Gamma\vdash\Delta,\mathcal{B}}{\Gamma\vdash\mathcal{A}\Rightarrow\mathcal{B},\Delta}\Rightarrow R$$

$$\frac{\mathcal{A}[x\leftarrow t],\Gamma\vdash\Delta}{\forall x.\mathcal{A},\Gamma\vdash\Delta}\forall L \qquad \frac{\Gamma\vdash\Delta,\mathcal{A}[x\leftarrow y]}{\Gamma\vdash\forall x.\mathcal{A},\Delta}\forall R \qquad \frac{\mathcal{A}[x\leftarrow y],\Gamma\vdash\Delta}{\exists x.\mathcal{A},\Gamma\vdash\Delta}\exists L \qquad \frac{\Gamma\vdash\Delta,\mathcal{A}[x\leftarrow t]}{\Gamma\vdash\exists x.\mathcal{A},\Delta}\exists R$$

per cui un sequente corrisponderebbe all'implicazione fra la congiunzione degli antecedenti e la disgiunzione dei conseguenti. *Questa corrispondenza non è formalmente corretta* in quanto un sequente, come già osservato, non appartiene ad un linguaggio logico (p.es., la logica dei predicati o la logica proposizionale), ma l'analogia può aiutare a ricordare certe regole del calcolo dei sequenti.

Un sequente è dimostrato se vale almeno una di queste condizioni:

- almeno un antecedente è falso;
- almeno un conseguente è vero;
- almeno una formula compare sia come antecedente che come conseguente (segue dalle due condizioni precedenti).

Il calcolo dei sequenti ha un solo assioma:

$$\Gamma,\mathcal{A}\vdash\mathcal{A},\Delta\ ,$$

dove Γ e Δ sono multiinsiemi di formule. Questo assioma si può giustificare informalmente in base alla corrispondenza vista sopra: considerando l'assioma come equivalente a $\Gamma\wedge\mathcal{A}\Rightarrow\mathcal{A}\vee\Delta$, si verifica che tale formula è una tautologia.

Le regole d'inferenza sono mostrate nella Tabella 4.5.5. Nella tabella, i simboli immediatamente a destra di ciascuna regola d'inferenza servono a identificare sinteticamente la regola. Per esempio, il simbolo '$\neg L$' si può leggere come "*inserimento a sinistra* (L, *left*) *della negazione*". Infatti la regola corrispondente dice che da un sequente della forma $\Gamma\vdash\Delta,\mathcal{A}$ si può dedurre un sequente della forma $\neg\mathcal{A},\Gamma\vdash\Delta$; analogamente per la regola etichettata dal simbolo '$\neg R$' (R sta per *right*). In base a queste due regole, qualsiasi formula si può spostare da un lato all'altro del simbolo di sequente, negandola (analogamente ai termini additivi di un'equazione algebrica).

Le etichette '*axm*', '*cut*' e '*ctr*' si possono leggere come '*assioma*', '*taglio*' e '*contrazione*'. L'unico assioma del sistema formale viene visto come una regola di inferenza con un insieme vuoto di premesse.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathcal{A},\mathcal{B}\vdash\mathcal{A}}\,\text{axm}}{\neg\mathcal{A},\mathcal{A},\mathcal{B}\vdash}\,\neg L \qquad \dfrac{}{\mathcal{A},\mathcal{B}\vdash\mathcal{B}}\,\text{axm}}{\neg\mathcal{B},\mathcal{A},\mathcal{B}\vdash}\,\neg L}{(\neg\mathcal{A}\vee\neg\mathcal{B}),\mathcal{A},\mathcal{B}\vdash}\,\vee L}{(\neg\mathcal{A}\vee\neg\mathcal{B}),(\mathcal{A}\wedge\mathcal{B})\vdash}\,\wedge L}{(\neg\mathcal{A}\vee\neg\mathcal{B})\vdash\neg(\mathcal{A}\wedge\mathcal{B})}\,\neg R}{\vdash(\neg\mathcal{A}\vee\neg\mathcal{B})\Rightarrow\neg(\mathcal{A}\wedge\mathcal{B})}\,\Rightarrow R$$

Figura 4.7: Una dimostrazione nel calcolo dei sequenti.

Nel calcolo dei sequenti, la dimostrazione di una formula $\mathcal{F}$ si costruisce all'indietro, partendo da un sequente della forma $\vdash\mathcal{F}$. Le regole d'inferenza vengono applicate all'indietro: dato un sequente, si cerca una regola la cui conseguenza abbia le stessa struttura del sequente, e questo viene sostituito dalle premesse, applicando le necessarie sostituzioni. Poiché alcune regole d'inferenza hanno due premesse, questo procedimento costruisce un albero di sequenti, chiamato *albero di dimostrazione* (*proof tree*), avente per radice il sequente iniziale. Applicando le varie regole, ogni ramo dell'albero può dare origine a nuovi rami, ma quando un sequente ha la stessa struttura dell'assioma ($\Gamma,\mathcal{A}\vdash\mathcal{A},\Delta$) la regola *axm* produce un sequente vuoto. La dimostrazione termina con successo se e quando tutti i rami sono chiusi con l'assioma. La fig. 4.7 mostra come esempio la dimostrazione della formula $\neg\mathcal{A}\vee\neg\mathcal{B}\Rightarrow\neg(\mathcal{A}\wedge\mathcal{B})$.

4.5.6 Logiche tipate

Nelle logiche tipate, il dominio è ripartito in *tipi* (*types*, *sorts*). Esiste un quantificatore per ciascun tipo, per ogni predicato viene fissato il tipo di ciascun argomento, e per ogni funzione vengono fissati i tipi degli argomenti e del suo valore. Le logiche tipate sono equivalenti alle logiche non tipate, nel senso che qualsiasi espressione di una logica tipata può essere tradotta in un'espressione non tipata equivalente, ma permettono di esprimere le specifiche in modo piú naturale, e soprattutto permettono di verificare, anche automaticamente, la correttezza delle espressioni, analogamente a quanto avviene con i linguaggi di programmazione tipati.

4.5.7 Logiche di ordine superiore

Nella logica del primo ordine le variabili possono rappresentare solo entità individuali, non funzioni, relazioni o insiemi. Quindi nella logica del primo ordine si possono esprimere delle frasi come "*per ogni x reale, $x^2 = x\cdot x$*", mentre non si possono esprimere delle frasi come "*per ogni funzione f di un numero reale, $f^2(x) = f(x)\cdot f(x)$*".

Nelle logiche di ordine superiore, le variabili possono rappresentare anche funzioni e relazioni, permettendo cosí di esprimere frasi come "*se x e y sono numeri reali e $x = y$,*

allora per ogni P tale che P sia un predicato unario si ha $P(x) = P(y)$". Generalmente le logiche di ordine superiore sono tipate.

4.5.8 Dimostrazione di teoremi assistita da calcolatore

Le proprietà di un sistema specificato con un linguaggio formale possono essere verificate con tecniche di *verifica assistita da calcolatore*, divise in due categorie: *dimostrazione di teoremi* (*theorem proving*) e *model checking*.

La dimostrazione di teoremi sfrutta dei software (*dimostratori di teoremi*) che implementano dei sistemi formali. Un dimostratore di teoremi accetta in ingresso una specifica formale del sistema e dei teoremi che esprimono le proprietà da verificare, e cerca di costruire delle dimostrazioni seguendo una strategia interamente automatica o guidata dallo sviluppatore.

Il *model checking* sfrutta dei software (*model checker*) che estraggono un *modello* del sistema dalla sua specifica formale. Il modello è un grafo i cui nodi sono gli stati del sistema, connessi dalle transizioni. Le proprietà del sistema vengono rappresentate come teoremi di una *logica temporale* (v. oltre), espressi in termini di stati (cioè di valori che caratterizzano gli stati, p.es. pressione e temperatura di un impianto) e di percorsi nel grafo (cioè sequenze di stati e transizioni). Il model checker esplora il grafo verificando se le proprietà richieste valgono per ogni stato e per ogni percorso.

In questa sezione tratteremo la dimostrazione di teoremi per mezzo del *Prototype Verification System* (PVS), un dimostratore interattivo di teoremi sviluppato al Computer Science Laboratory dell'SRI International [35]. L'ambiente PVS è disponibile gratuitamente dal sito `http://pvs.csl.sri.com`.

Introduzione

Il sistema formale del PVS si basa su un linguaggio tipato di ordine superiore e sul calcolo dei sequenti. È stato applicato in molti campi, fra cui la verifica formale di sistemi digitali, algoritmi, e sistemi safety-critical.

La verifica delle proprietà di un sistema si svolge come segue:

- si descrive il sistema per mezzo di una teoria, che comprende definizioni di tipi, variabili e funzioni, e gli assiomi richiesti;
- si scrivono le formule che rappresentano le proprietà da dimostrare;
- si seleziona una di tali formule e si entra nell'ambiente di dimostrazione interattiva;
- in tale ambiente si usano dei comandi che applicano le regole di inferenza del calcolo dei sequenti, trasformando il sequente iniziale fino ad ottenere la dimostrazione (se possibile).

La dimostrazione, quindi, non viene eseguita automaticamente dallo strumento, ma viene guidata dallo sviluppatore, che ad ogni passo sceglie il comando da applicare. Cia-

scun comando, però, può applicare una combinazione di piú regole di inferenza o applicarle ripetutamente, per cui una dimostrazione complessa si può spesso risolvere in pochi passi[9].

Un esempio

Il seguente esempio mostra una semplice teoria sulla struttura algebrica dei gruppi, in cui si vuole dimostrare una proprietà della funzione *inverso*. Osserviamo che nell'ambiente PVS tutte le formule sono chiuse: se una variabile non ha un quantificatore esplicito, viene considerata come quantificata universalmente.

```
group : THEORY
BEGIN
  G : TYPE+      % insieme non interpretato, non vuoto
  e : G          % elemento neutro
  i : [G -> G]   % inverso
  * : [G,G -> G] % operazione binaria da G x G a G
  x,y,z : VAR G
  associative : AXIOM
    (x * y) * z = x * (y * z)
  id_left : AXIOM
    e * x = x
  inverse_left : AXIOM
    i(x) * x = e
  inverse_associative : THEOREM
    i(x) * (x * y) = y
END group
```

Questa teoria descrive una struttura algebrica formata dall'insieme G e dall'operazione binaria '$*$' (che chiameremo, per semplicità, "prodotto") che gode delle proprietà di chiusura, associatività, esistenza dell'elemento neutro e invertibilità.

La prima dichiarazione dice che G è un tipo *non interpretato*, cioè non definito in termini di altri tipi, e non vuoto (indicato dal simbolo '+'). L'elemento neutro e (*Einselement*) appartiene a G. Seguono le dichiarazioni dell'operazione di inversione i e dell'operazione binaria caratteristica del gruppo, e le variabili x, y e z che assumono valori in G.

I tre assiomi definiscono la proprietà di associatività, l'elemento neutro (o identico) e la proprietà di invertibilità. Le variabili che compaiono in queste formule sono quantificate universalmente per default.

Infine, un teorema da dimostrare: *per ogni x e y appartenenti a G, il prodotto dell'inverso di x per il prodotto di x ed y è uguale a y.*

Per dimostrare il teorema, si seleziona lo stesso col cursore e si attiva l'ambiente di dimostrazione (*dimostratore*, *prover*). Viene mostrato il sequente iniziale, o *goal*:

[9]Peraltro, può succedere che un passaggio intuitivamente banale si traduca in un certo numero di passi di deduzione formale.

```
inverse_associative :

  |-------
{1}   FORALL (x, y: G): i(x) * (x * y) = y

Rule?
```

dove la barra verticale seguita dalla linea tratteggiata rappresenta il simbolo di sequente, e il goal ha quindi la forma $\vdash \forall x \forall y(i(x) * (x * y)) = y$. Il prompt '`Rule?`' chiede allo sviluppatore di scegliere una regola.

Si inseriscono gli assiomi nell'antecedente, applicando la regola *lemma*:

```
Rule? (lemma "associative")
{-1}  FORALL (x, y, z: G): (x * y) * z = x * (y * z)
  |-------
[1]   FORALL (x, y: G): i(x) * (x * y) = y

Rule? (lemma "inverse_left")
   ...
Rule? (lemma "id_left")
   ...
```

Nel conseguente, si inseriscono delle costanti arbitrarie (dette *di Skolem*) per eliminare il quantificatore dal conseguente, applicando la regola *skosimp* (skolemizzare e semplificare). Per default, gli identificatori di queste costanti sono formate dal nome della variabile da eliminare seguite da un punto esclamativo e da un numero.

```
Rule? (skosimp*)
[-1]  FORALL (x: G): e * x = x
[-2]  FORALL (x: G): i(x) * x = e
[-3]  FORALL (x, y, z: G): (x * y) * z = x * (y * z)
  |-------
{1}   i(x!1) * (x!1 * y!1) = y!1
```

Si sostituiscono le costanti di Skolem nelle formule dell'antecedente, eliminando i quantificatori, con la regola *inst*:

```
Rule? (inst -3 "i(x!1)" "x!1" "y!1")
   ...
Rule? (inst -2 "x!1")
   ...
Rule? (inst -1 "x!1")
{-1}  e * x!1 = x!1
[-2]  i(x!1) * x!1 = e
[-3]  (i(x!1) * x!1) * y!1 = i(x!1) * (x!1 * y!1)
  |-------
[1]   i(x!1) * (x!1 * y!1) = y!1
```

Si applicano delle semplificazioni secondo l'algebra booleana, con la regola *grind*:

```
Rule? (grind)
[-1]  e * x!1 = x!1
[-2]  i(x!1) * x!1 = e
{-3}  e * y!1 = i(x!1) * (x!1 * y!1)
  |-------
[1]   i(x!1) * (x!1 * y!1) = y!1
```

Si usa la formula (-3) dell'antecedente per semplificare il conseguente, sostituendovi il primo membro dell'uguaglianza con la regola *replace*:

```
Rule? (replace -3 :dir RL)
[-1]  e * x!1 = x!1
[-2]  i(x!1) * x!1 = e
[-3]  e * y!1 = i(x!1) * (x!1 * y!1)
  |-------
{1}   e * y!1 = y!1
```

Si reinserisce l'assioma dell'identità:

```
Rule? (lemma "id_left")
{-1}  FORALL (x: G): e * x = x
[-2]  e * x!1 = x!1
[-3]  i(x!1) * x!1 = e
[-4]  e * y!1 = i(x!1) * (x!1 * y!1)
  |-------
[1]   e * y!1 = y!1
```

E infine si sostituisce la costante di Skolem del conseguente nell'assioma dell'identità, concludendo la dimostrazione:

```
Rule? (inst -1 "y!1")

Q.E.D.
```

Il linguaggio

Il linguaggio di specifica del PVS comprende vari connettivi logici, come `NOT`, `AND`, `OR`, `IMPLIES`. Le espressioni sia booleane che aritmetiche possono usare, oltre ai rispettivi connettivi ed operatori, gli operatori complessi come `IF-THEN-ELSE` e `COND` (una combinazione di `IF-THEN-ELSE` analoga alle istruzioni *switch* o *case* dei linguaggi imperativi). I quantificatori sono `EXISTS` e `FORALL`.

Tipi Il sistema dei tipi comprende tipi base predefiniti, come `bool`, `nat` e `real`. Come mostrato nell'esempio precedente, dichiarare un tipo *non interpretato* in una teoria significa semplicemente che esiste un tipo con un certo nome, senza dare ulteriori indicazioni sui suoi possibili elementi. Questo non impedisce di usarlo nella teoria, dichiarando variabili e costanti di quel tipo.

I tipi *interpretati* ricadono in diverse categorie, fra cui:

enumerazioni: analoghe ai tipi enumerati del C++, per esempio
`flag: TYPE = {red, black, white, green}`.

n-uple: n-uple ordinate di valori, anche di tipi diversi, per esempio
`triple: TYPE = [nat, flag, real]`.

record: analoghe ai tipi struttura del C++, per esempio
`point: TYPE = [# x: real, y: real #]`.

sottotipi: restrizioni di altri tipi, definite da vincoli sugli elementi del tipo originale, come in
`posnat: TYPE = {x: nat | x > 0}`.

funzioni: definiti da dominio e codominio, come in
`int2int: TYPE = [int -> int]`.

Costanti e variabili Le costanti si dichiarano come nei seguenti esempi, e le variabili si dichiarano allo stesso modo, inserendo la parola `VAR` prima dell'espressione di tipo:

- `n0: nat` (*costante non interpretata*)
- `lucky: nat = 13`
- `a_triple: triple = (lucky, red, 3.14)`
- `t1: VAR triple`
- `t2: VAR [nat, flag, real]`.
- `origin: point = (# x := 0.0, y:= 0.0 #)`

Le *costanti funzione* (da non confondere con le funzioni costanti) si possono dichiarare in diversi modi. L'esempio seguente mostra i modi in cui si può dichiarare la funzione `inc` che incrementa di uno un valore intero:

- `inc: int2int = (lambda (x: int): x + 1)`
- `inc: [int -> int] = (lambda (x: int): x + 1)`
- `inc(x: int): int = x + 1`

La prima dichiarazione dice che `inc` è una costante funzione di tipo `int2int` il cui valore è l'espressione `(lambda (x: int): x + 1)`. Questa è una λ-*espressione*, un meccanismo che permette di rappresentare la legge che associa i valori degli argomenti (elementi del dominio) al valore della funzione (elemento del codominio) astraendo dal nome della funzione, creando cosí una *funzione anonima*. Usando il lessico dei linguaggi imperativi, si può dire che una λ-espressione è il "corpo" di una funzione.

Nella seconda dichiarazione è come la prima, salvo che il tipo viene definito localmente, senza il bisogno di una precedente dichiarazione di tipo.

Infine, la terza dichiarazione usa una sintassi simile a quelle usate nei linguaggi di programmazione, che l'ambiente PVS traduce internamente nella sintassi delle dichiarazioni precedenti, usando λ-espressioni.

Formule Le formule sono espressioni logiche identificate da un nome e classificate come *assiomi* o *teoremi*. Gli assiomi vengono dichiarati con la parola `AXIOM` ed i teoremi con la parola `THEOREM` o suoi sinonimi (p.es. `LEMMA`) come nel seguente esempio:

- `plus_commutativity: AXIOM`
  `forall(x, y: nat): x + y = y + x`
- `a_theorem: THEOREM forall(n: nat): n < n + 1`

Dimostrazioni

Il dimostratore interattivo costruisce un albero di dimostrazione eseguendo *comandi* (o *regole*), divisi in varie categorie:

controllo: controllo dell'esecuzione ed esplorazione dell'albero di dimostrazione.
strutturali: per implementare le regole di contrazione (*supra*, sez. 4.5.5) e nascondere formule non usate in un sequente.
proposizionali: implementano le regole d'inferenza per i connettivi e gli operatori complessi, e la regola del taglio o *cut* (*supra*, sez. 4.5.5), oltre ad applicare varie regole di semplificazione.
quantificazione: implementano le regole d'inferenza per i quantificatori.
uguaglianza: implementano regole d'inferenza per l'uguaglianza, per espressioni contenenti record o n-uple, e definizioni di funzioni.
definizioni e lemmi: inseriscono ed usano nei sequenti lemmi e definizioni.
strategie: applicano sequenze prestabilite di comandi.

Un esempio di verifica

Consideriamo un semplice caso per mostrare come un sistema tecnico, specificamente un circuito logico, si possa modellare in PVS.

Un semiaddizionatore (*half adder*, *HA*) è un circuito digitale i cui due ingressi rappresentano la codifica su una cifra binaria di due numeri naturali, e la cui uscita è la codifica binaria su due cifre della loro somma. La cifra meno significativa viene chiamata *sum* (somma) e la piú significativa *carry* (riporto). La fig. 4.8 ne mostra un'implementazione, di cui vogliamo verificare la correttezza. La seguente teoria contiene il modello del circuito e il teorema di correttezza da dimostrare:

```
HA : THEORY
BEGIN
  x,y : VAR bool
```

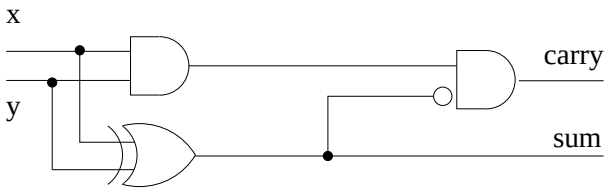


Figura 4.8: Un'implementazione del semiaddizionatore.

```
  HA(x,y) : [bool, bool] =
     ((x AND y) AND (NOT (x XOR y)), % carry
      (x XOR y))                     % sum

  % conversione da booleano (T, F) a naturale (0, 1)
  b2n(x) : nat = IF x THEN 1 ELSE 0 ENDIF

  HA_corr : THEOREM                  % correttezza
    LET (carry, sum) = HA(x, y) IN
      2*b2n(carry) + b2n(sum) = b2n(x) + b2n(y)
END HA
```

La funzione *HA* restituisce una coppia ordinata di valori booleani corrispondenti al riporto e alla somma, calcolati per mezzo delle espressioni booleane implementate dal circuito. La funzione *b2n* converte un valore booleano in un numero naturale.

Nel teorema di correttezza, l'espressione *let* pone la coppia di variabili $(carry, sum)$ uguale al risultato della funzione *HA*. Il primo membro dell'equazione successiva calcola il numero naturale rappresentato da *carry* e *sum*, ed il secondo è la somma dei numeri naturali codificati da x e y.

In modo analogo si può dimostrare la correttezza di un'implementazione alternativa, o l'equivalenza di due implementazioni[10].

4.5.9 Logiche modali e temporali

Le logiche modali arricchiscono il linguaggio della logica permettendo di esprimere aspetti del ragionamento che nel linguaggio naturale possono essere resi dai modi grammaticali (indicativo, congiuntivo, condizionale). Per esempio, la frase "se piovesse non uscirei" suggerisce che non sta piovendo nel momento in cui viene pronunciata, *oltre* ad esprimere il fatto che la pioggia e l'azione di uscire sono legate da un'implicazione; questa sovrapposizione di significati non si ha nella frase "se piove non esco". In particolare, una logica modale può distinguere una verità necessaria da una verità contingente. Le logiche temporali sono una classe di logiche modali, e servono ad esprimere il fatto che certe formule sono vere o false a seconda dell'intervallo di tempo o dell'istante in cui si valutano.

[10] `http://www.ing.unipi.it/~a009435/issw/isw2122_slides.pdf`

Le logiche modali presuppongono che le loro formule vengano valutate rispetto a piú "mondi possibili" (universi di discorso), a differenza della logica non modale che si riferisce ad un solo mondo. Il fatto che una formula sia necessariamente vera corrisponde, per la semantica delle logiche modali, al fatto che la formula è vera in tutti i mondi considerati possibili in una particolare logica.

La sintassi delle logiche modali è quella delle logiche non modali, a cui si aggiungono *operatori modali*, di cui i principali sono:

- operatore di necessità $\Box$;
- operatore di possibilità $\Diamond$,

legati dalle relazioni

$$\Box\mathcal{F} \Leftrightarrow \neg\Diamond\neg F$$
$$\Diamond\mathcal{F} \Leftrightarrow \neg\Box\neg F$$

L'insieme $\mathcal{W}$ delle formule ben formate viene quindi esteso con questa regola:

- se $\mathcal{F}$ è una formula, allora sono formule anche $\Box\mathcal{F}$ e $\Diamond\mathcal{F}$.

La semantica di una logica modale è data da una *terna di Kripke* $\langle \mathbf{W}, \mathcal{R}, V \rangle$, dove:

- l'insieme $\mathbf{W}$ è l'insieme dei mondi (o interpretazioni);
- $\mathcal{R}$ è la *funzione di accessibilità* : $\mathcal{R} : \mathbf{W} \to \mathbf{W}$;
- V è la *funzione di valutazione* $V : \mathcal{W} \times \mathbf{W} \to \mathbb{B}$;

La funzione di accessibilità (o *visibilità*) assegna una struttura all'insieme dei mondi possibili, determinando quali mondi sono considerati possibili a partire dal "mondo attuale": p.es., in una logica temporale, i mondi possibili in un dato istante possono essere quelli associati agli istanti successivi. La funzione di accessibilità è riflessiva e transitiva.

La funzione di valutazione è tale che:

- $V(\Box\mathcal{F}, w) = \mathbf{T}$ se e solo se, per ogni $v \in \mathbf{W}$ tale che $\mathcal{R}(w, v)$, si ha $V(\mathcal{F}, v) = \mathbf{T}$;
- $V(\Diamond\mathcal{F}, w) = \mathbf{T}$ se e solo se esiste un $v \in \mathbf{W}$ tale che $\mathcal{R}(w, v)$ e $V(\mathcal{F}, v) = \mathbf{T}$;

Seguono alcuni assiomi per la logica modale:

$$\Box\neg\mathcal{P} \Leftrightarrow \neg\Diamond\mathcal{P} \tag{4.9}$$
$$\Box(\mathcal{P} \Rightarrow \mathcal{Q}) \Rightarrow (\Box\mathcal{P} \Rightarrow \Box\mathcal{Q}) \tag{4.10}$$
$$\Box\mathcal{P} \Rightarrow \mathcal{P} \tag{4.11}$$

I tre schemi di assiomi si possono leggere, rispettivamente, cosí:

- (4.9): $\mathcal{P}$ è necessariamente falso se e solo se $\mathcal{P}$ non può essere vero.
- (4.10): se $\mathcal{P}$ implica necessariamente $\mathcal{Q}$, allora la necessità di $\mathcal{P}$ implica la necessità di $\mathcal{Q}$.
- (4.10): la necessità di $\mathcal{P}$ implica $\mathcal{P}$.

Logica temporale

Le logiche temporali servono a ragionare su come un sistema si evolve nel tempo, permettono cioè di esprimere affermazioni come "la temperatura del refrigerante non supererà *mai* il limite fissato" o come "se i segnali A e B vengono attivati, *prima o poi* verrà attivato il segnale C". In una logica temporale, lo stato di un sistema è definito in ogni istante dai valori di un insieme di *variabili di stato* che possono rappresentare quantità fisiche o condizioni logiche, e le proprietà del sistema in ogni stato vengono espresse da formule di una logica soggiacente, come il calcolo proposizionale o quello dei predicati. Queste formule vengono poi combinate per mezzo di operatori temporali che ne mettono i valori di verità in relazione a istanti o intervalli di tempo.

Nella logica temporale, i mondi possibili rappresentano quindi *stati* del sistema su cui vogliamo ragionare, corrispondenti a diversi istanti del tempo. La relazione di accessibilità descrive la struttura del tempo: per esempio, se ammettiamo che ad uno stato possano seguire due o piú stati alternativi, abbiamo un tempo ramificato. Si possono quindi considerare dei tempi lineari, o ramificati, o circolari (sistemi periodici), o ancora piú complessi. Inoltre il tempo può essere continuo o discreto.

La Linear Temporal Logic LTL La *Linear Temporal Logic* (LTL) presuppone un sistema di riferimento temporale di tipo lineare e discreto. L'insieme dei mondi possibili è quindi una successione di stati s_i, ove la relazione di accessibilità definisce le transizioni possibili ed è tale che in ogni istante lo stato del sistema abbia un solo successore possibile.

Nella LTL i due operatori modali principali assumono questi significati:
- $\Box$ (*henceforth*) oppure **G** (*globally*), d'ora in poi;
- $\Diamond$ (*eventually*) oppure **F** (*in the future*), prima o poi;

Da questi operatori se ne possono definire altri:
- **U** *until*, finché ($p\mathbf{U}q$ se e solo se esiste k tale che q è vero in s_k e p è vero per tutti gli s_i con $i \leq k$);
- $\blacksquare$ *always in the past*, sempre in passato;
- $\blacklozenge$ *sometime in the past*, qualche volta in passato;
- $\circ$ *next*, prossimo istante (per un riferimento temporale discreto).

Nell'esempio seguente usiamo la LTL per modellare un semplice sistema a scambio di messaggi fra due processi A e B che possono spedire e ricevere messaggi m e spedire un messaggio di acknowledgement in risposta alla ricezione. La specifica del relativo protocollo di comunicazione potrebbe contenere il seguente frammento:

$$\Box(\text{send(m)} \Rightarrow \text{state} = \textbf{connected})$$
$$\Box(\text{send\_ack(m)} \Rightarrow \blacklozenge\ \text{receive(m)})$$
$$\Box(\text{B.receive(m)} \Rightarrow \blacklozenge\ \text{A.send(m)})$$
$$\Box(\text{A.send(m)} \Rightarrow \Diamond \text{B.receive(m)})$$
$$\Box(\Box\Diamond \text{A.send(m)} \Rightarrow \Diamond \text{B.receive(m)})$$
$$\Box(\text{B.receive(m)} \Rightarrow \Diamond \text{B.send\_ack(m)})$$

Le prime tre formule esprimono proprietà di *sicurezza* (non succede niente di brutto):

- in ogni stato, se viene spedito un messaggio m, la connessione è attiva;
- in ogni stato, se viene mandato un acknowledgement per un messaggio, il messaggio è già stato ricevuto;
- in ogni stato, se il processo B riceve un messaggio, il messaggio è già stato spedito;

Evidentemente, la falsità di una delle formule precedenti segnalerebbe un malfunzionamento piuttosto grave.

Le formule successive esprimono proprietà di *vitalità* (prima o poi succede qualcosa di buono):

- in ogni stato, se viene spedito un messaggio m, prima o poi viene ricevuto;
- in ogni stato, se un messaggio m viene spedito piú volte, prima o poi viene ricevuto;
- in ogni stato, se il processo B riceve un messaggio, prima o poi restituisce un acknowledgement.

La Computation Tree Logic Per modellare evoluzioni alternative (*rami*, *cammini*, *path*, *tracce* ...) di un sistema, la CTL introduce gli operatori

- **A**: lungo tutti i cammini;
- **E**: lungo almeno un cammino.

Questi operatori si combinano con quelli della LTL:

- **AG**(p): per ogni cammino, p è sempre vero;
- **AF**(p): per ogni cammino, p prima o poi è vero;
- **EG**(p): esiste almeno un cammino in cui p è sempre vero;
- **EF**(p): esiste almeno un cammino in cui p prima o poi è vero.
- ...

Semantica Generalmente, per analizzare un sistema usando logiche temporali, il sistema viene modellato come una *struttura di Kripke*, una macchina a stati piú generale degli automi a stati finiti. Una struttura di Kripke si ottiene aggiungendo alla terna $\langle \mathbf{W}, \mathcal{R}, V \rangle$ una funzione (di *labeling*) che associa ad ogni stato le formule logiche che esprimono le proprietà del sistema. Tali strutture possono essere descritte per mezzo di linguaggi *orientati agli stati*, nei quali un sistema viene descritto per mezzo delle variabili che ne definiscono lo stato, e delle transizioni possibili. Avendo modellato un sistema in forma di struttura di Kripke, se ne possono specificare le proprietà con una logica temporale e verificarle usando strumenti basati sul model-checking.

Il model checking

Come anticipato, una formula di una logica temporale viene comunemente verificata col *model checking*. Un model checker costruisce la struttura di Kripke (o *spazio degli stati*) di un sistema specificato in opportuno formalismo e vàluta la formula sui percorsi possibili. Se la formula non viene soddisfatta, il model checker di solito produce un *controesempio*, cioè un percorso nello spazio degli stati che conduce ad un stato in cui la formula non vale.

Nel séguito verrà introdotto sommariamente il SAL (*Symbolic Analysis Laboratory*[11]), uno strumento per il model checking sviluppato al Computer Science Laboratory dello Stanford Research Institute International.

Il SAL ha un linguaggio per definire sistemi di transizioni e può verificare formule in LTL e (limitatamente) in CTL.

Lo spazio degli stati di un sistema semplice viene definito da un *modulo* il cui stato è definito dall'insieme dei valori delle variabili di ingresso, di uscita e locali in esso dichiarate. Per le variabili locali e di uscita sono definiti il *valore attuale*, denotato dal nome della variabile, p.es. *X*, ed il *valore successivo*, denotato dal nome della variabile con apice, p.es. *X'*. Le variabili vengono inizializzate nella sezione `INITIALIZATION` del modulo.

Le transizioni sono definite da assegnamenti al valore successivo delle variabili, nella sezione `TRANSITIONS`.

Lo spazio degli stati di un sistema complesso viene definito combinando piú moduli per mezzo di operatori di sincronizzazione.

L'esempio seguente mostra come si può modellare in SAL un semplice sistema che si può essere libero (*ready*) o occupato (*busy*). Se è libero e riceve una richiesta diventa occupato, altrimenti può restare o diventare libero oppure occupato, nondeterministicamente. Il codice SAL è un *contesto* (analogo ad una teoria del PVS, sez. 4.5.8) contenente la dichiarazione del tipo `State`, il modulo `main` ed una proprietà da dimostrare nell'àmbito della LTL (teorema *th1*) e della CTL (*th2*). La proprietà è il requisito che, se in qualsiasi istante (e in qualsiasi ramo dell'albero di computazione, nel teorema *th2*) si riceve una richiesta, in un istante successivo il sistema entri nello stato *busy*.

Nel modulo `main`, lo stato è definito dalle variabili `state` di tipo `State` e `request` di tipo `boolean`. Quest'ultima rappresenta l'arrivo di una richiesta, modellato dal valore *vero*. La variabile `state` viene inizializzata a `ready` e le possibili transizioni sono definite calcolando il valore successivo di `state` secondo la regola esposta nel paragrafo procedente. L'operatore `IN` rappresenta l'assegnamento nondeterministico.

```
short: CONTEXT = BEGIN
  State: TYPE = {ready, busy};
  main: MODULE = BEGIN
      INPUT request: boolean          % TRUE se c'e' una richiesta
```

[11] `http://sal.csl.sri.com`

```
      OUTPUT state: State
      INITIALIZATION state = ready
      TRANSITION
        state'                          % valore successivo
        IN                              % scelta nondeterministica
        IF (state = ready) AND request
        THEN {busy}
        ELSE {ready, busy}           % insieme di scelte possibili
        ENDIF
    END;
th1: THEOREM main |- G(request => F(state = busy));    % LTL
th2: THEOREM main |- AG(request => AF(state = busy));  % CTL
END
```

4.6 Linguaggi orientati agli oggetti

In questa sezione vengono esposti i concetti fondamentali dell'analisi e specifica orientata agli oggetti, facendo riferimento ad un particolare linguaggio di specifica, l'UML (*Unified Modeling Language*).

Nell'analisi orientata agli oggetti, la descrizione di un sistema parte dall'identificazione di *oggetti* (cioè elementi costitutivi) e di *relazioni* fra oggetti.

Un oggetto rappresenta un'entità individuale nel dominio di applicazione del sistema da sviluppare, ed è definito dalla propria *identità*, dai propri *attributi*, cioè da un'insieme di proprietà che lo caratterizzano, e dalle *operazioni* che l'oggetto può compiere interagendo con altri oggetti; queste operazioni possono avere dei parametri di ingresso forniti dall'oggetto che *chiama* (o *invoca*) un'operazione, e restituire risultati in funzione dei parametri d'ingresso e degli attributi dell'oggetto che *esegue* l'operazione.

Gli attributi, oltre a rappresentare proprietà degli oggetti, possono rappresentarne lo *stato*, che può essere modificato dalle operazioni. Per chiarire la differenza fra *proprietà* e *stato*, possiamo pensare di rappresentare un'automobile con due attributi, vel_max (velocità massima) e marcia: il primo rappresenta una proprietà fissa, mentre il secondo rappresenta i diversi stati di funzionamento della trasmissione (potrebbe assumere i valori *retromarcia*, *folle*, *prima*...). Possiamo completare la descrizione dell'automobile con le operazioni imposta_vel_max(), marcia_sup() e marcia_inf(): la prima permette di impostare o cambiare il valore di una proprietà (per esempio dopo una modifica meccanica), le altre a cambiare lo stato di funzionamento. Quindi il modello orientato agli oggetti unifica i tre punti di vista dei linguaggi di specifica: la descrizione dei dati, delle funzioni, e del controllo. Quest'ultimo aspetto, come vedremo, viene modellato con maggior completezza e precisione anche con altri linguaggi, complementari a quello orientato agli oggetti, fra cui quello basato sulle macchine a stati.

Generalmente, in un sistema esistono piú oggetti simili, cioè delle entità distinte che hanno gli stessi attributi (con valori eventualmente, ma non necessariamente, diversi) e

le stesse operazioni. Una *classe* è una descrizione della struttura e del comportamento comuni a piú oggetti. Una specifica orientata agli oggetti consiste essenzialmente in un insieme di classi e di relazioni fra classi.

Un'altra caratteristica del modello orientato agli oggetti è il ruolo che ha in esso il concetto di *generalizzazione*. Questo concetto permette di rappresentare il fatto che alcune classi hanno in comune una parte della loro struttura o del loro comportamento.

I concetti fondamentali del modello orientato agli oggetti si possono cosí riassumere:

oggetti: un oggetto corrisponde ad un'entità individuale del sistema che vogliamo modellare. Gli oggetti hanno *attributi*, che ne definiscono le proprietà o lo stato, e *operazioni*, che ne definiscono il comportamento. Inoltre, ogni oggetto ha una *identità* che permette di distinguerlo dagli altri oggetti[12].

legami: oggetti distinti possono essere in qualche relazione fra loro: tali relazioni fra oggetti sono dette *legami* (*link*). Un legame si può rappresentare formalmente come la coppia dei due oggetti collegati.

classi: gli oggetti che hanno la stessa struttura e comportamento sono raggruppati in classi, ed ogni oggetto è un'*istanza* di qualche classe. Una classe quindi descrive un insieme di oggetti che hanno la stessa struttura (cioè lo stesso insieme di attributi) e lo stesso comportamento, ma sono distinguibili l'uno dall'altro, e in generale, ma non necessariamente, hanno diversi valori degli attributi.

associazioni: un'associazione sta ad un legame come una classe sta ad un oggetto, è un insieme di link simili per struttura e significato. Definendo come struttura di un link la coppia dei tipi dei due oggetti collegati, un'associazione binaria è definita dalle classi cui appartengono gli oggetti collegati da link con la stessa struttura. Piú in generale, un'associazione può rappresentare un insieme di strutture complesse di legami: per esempio, se dei viaggiatori hanno ciascuno un legame con la propria destinazione ed un legame con un mezzo di trasporto, questa situazione si può rappresentare con un'associazione ternaria fra le classi che rappresentano i viaggiatori, le destinazioni ed i mezzi di trasporto. Un'associazione quindi corrisponde formalmente ad una relazione n-aria.

altre relazioni: Si possono rappresentare altri tipi di relazioni, fra cui vari tipi di dipendenza e la generalizzazione. Osserviamo che questa è una relazione fra classi (corrispondente all'inclusione nella teoria degli insiemi), non fra oggetti.

4.6.1 L'UML

L'UML (*Unified Modeling Language*) [42] si basa su una notazione grafica orientata agli oggetti, applicabile dalla fase di analisi e specifica dei requisiti alla fase di codifica, anche se non vincola quest'ultima fase all'uso di linguaggi orientati agli oggetti: un sistema progettato in UML può essere implementato con linguaggi non orientati agli oggetti, sebbene questi ultimi, ovviamente, non siano la scelta piú naturale. Per il momento studieremo alcuni aspetti dell'UML relativi alla fase di analisi e specifica.

[12] Osserviamo però che in UML esistono anche i *tipi di dati astratti*, non trattati in questo corso, le cui istanze non hanno identità individuale, analogamente ai numeri.

Il linguaggio UML, la cui standardizzazione è curata dall'OMG (Object Management Group[13]), è passato attraverso una serie di revisioni. Al momento della pubblicazione di questa dispensa (2022), la versione piú recente è la 2.5.1 (2017). In questo corso si vuol dare soltanto una sintetica introduzione ai concetti fondamentali di questo linguaggio, rinunciando sia alla completezza che al rigore, per cui la maggior parte delle nozioni qui esposte è applicabile a qualsiasi versione dell'UML. Ove sia necessario mettere in evidenza qualche differenza di concetti, di notazioni o di terminologia, si userà il termine "UML2" per la versione attuale, e "UML1" per le versioni precedenti.

In UML, un sistema viene descritto da vari punti di vista, o *viste* (*views*). Per il momento ci occuperemo dei punti di vista piú rilevanti nella fase di analisi: (i) il punto di vista dei *casi d'uso* (*use cases*), che definisce i servizi forniti dal sistema agli utenti; (ii) il punto di vista *statico* (o *strutturale*), che ne descrive la struttura; e quello (iii) *dinamico*, che ne descrive il comportamento. Il punto di vista fondamentale è quello statico[14], cioè la descrizione delle classi e delle relazioni nel sistema modellato.

Per ciascun punto di vista, si costruiscono uno o piú modelli parziali del sistema, ciascuno dei quali può rappresentarne una parte o un aspetto particolare, e inoltre può contenere informazioni piú o meno dettagliate, secondo il livello di astrazione richiesto. Un modello è un insieme strutturato di informazioni organizzate in *elementi di modello*: l'insieme dei vari tipi di elementi di modello costituisce il "vocabolario" concettuale di cui si serve lo sviluppatore per definire i singoli modelli. Per esempio, in UML esistono gli elementi di tipo *classe* e *associazione* corrispondenti ai concetti esposti nelle sezioni precedenti. Gli *attributi* e le *operazioni*, a loro volta, sono tipi di elementi di modello che fanno parte degli elementi *classe*.

Gli elementi di modello, e quindi i modelli di cui fanno parte, rappresentano la semantica del sistema. Agli elementi di modello corrispondono *elementi di presentazione* che visualizzano gli elementi di modello in forma grafica e testuale. Generalmente, un elemento di presentazione può rappresentare, opzionalmente, solo una parte delle informazioni contenute nel corrispondente elemento di modello, cosí che il livello di dettaglio della rappresentazione grafica di un modello possa essere scelto di volta in volta secondo le esigenze. Molti elementi hanno una forma minima (per esempio, un semplice simbolo) e delle forme estese, piú ricche di informazioni. Per esempio, una classe può essere rappresentata da un rettangolo contenente solo il nome della classe, oppure da un rettangolo diviso in tre scompartimenti, contenenti nome, attributi ed operazioni, che a loro volta possono essere specificati in modo piú o meno dettagliato.

Un *diagramma* è un insieme di elementi di presentazione, che visualizza una parte di un modello. Esistono diversi tipi di diagrammi, ciascuno costituito da diversi tipi di elementi di presentazione, con le proprie regole di composizione. Ciascun elemento di modello può apparire in piú di un diagramma.

I diagrammi possono contenere delle *note*, cioè dei commenti, che si presentano come una raffigurazione stilizzata di un biglietto con un "orecchio" ripiegato, e possono essere

[13] `http://www.omg.org`

[14] In alcune metodologie, però, si dà un ruolo centrale al punto di vista dei casi d'uso.

collegate ad un elemento di presentazione con una linea tratteggiata.

In UML esistono tre meccanismi di estensione che permettono di adattare il linguaggio a esigenze specifiche: *vincoli*, *valori etichettati* e *stereotipi*. Questi meccanismi verranno trattati in séguito (sez. 4.6.10).

L'UML è un linguaggio *semiformale* in quanto permette di costruire modelli *incompleti*. Per esempio:

- la semantica delle operazioni può non essere definita;
- il tipo degli attributi può non essere definito;
- il tipo degli argomenti di un'operazione può non essere definito;
- una classe può essere definita senza specificarne attributi e operazioni.

Osserviamo che la *semantica* delle operazioni *può* essere definita in vari modi, mentre la loro *implementazione* non viene mai rappresentata esplicitamente in un modello UML.

Inoltre, distinguiamo l'incompletezza semantica di un elemento di modello dall'*elisione* di elementi di rappresentazione. Il termine "elisione" significa che dalla rappresentazione di qualsiasi elemento di modello si possono omettere, di volta in volta, i tratti non necessari in un dato contesto.

I modelli UML vengono costruiti per mezzo di strumenti *CASE* (*computer-aided software engineering*) dotati di editor grafici ed interfacce testuali. Generalmente questi strumenti permettono di verificare la correttezza sintattica dei diagrammi e, almeno in parte, la consistenza semantica del modello. Inoltre possono produrre documentazione e implementazioni parziali del modello, generando codice sorgente.

4.6.2 Classi e oggetti

Come abbiamo visto, una classe rappresenta astrattamente un insieme di oggetti che hanno gli stessi attributi, operazioni, relazioni e comportamento. In un modello di analisi una classe si usa per rappresentare un concetto pertinente al sistema che viene specificato, concetto che può essere concreto (per esempio, un controllore di dispositivi) o astratto (per esempio, una transazione finanziaria). In un modello di progetto una classe rappresenta un'entità software introdotta per implementare l'applicazione, per esempio una classe del linguaggio C++.

Una classe viene rappresentata graficamente da un rettangolo contenente il nome della classe e, opzionalmente, l'elenco degli attributi e delle operazioni. Se la classe ha uno o piú stereotipi (sez. 4.6.10), i loro nomi vengono scritti sopra al nome della classe. Uno stereotipo può anche essere rappresentato da un'icona nell'angolo destro in alto del rettangolo.

La rappresentazione minima di una classe consiste in un rettangolo contenente solo il nome ed eventualmente uno o piú stereotipi. Se la classe ha uno stereotipo rappresentabile da un'icona, la rappresentazione minima consiste nell'icona e nel nome (vedi, p.es., fig. 4.41).

Attributi

Ogni attributo ha un *nome*, che è l'unica informazione obbligatoria. Le altre informazioni associate agli attributi sono:

tipo: può essere uno dei tipi fondamentali predefiniti dall'UML (corrispondenti a quelli usati comunemente nei linguaggi di programmazione), un tipo definito in un linguaggio di programmazione, o una classe definita (in UML) dallo sviluppatore.

visibilità: *privata*, *protetta*, *pubblica*, *package*; quest'ultimo livello di visibilità significa che l'attributo è visibile da tutte le classi appartenenti allo stesso package (sez. 5.4.1).

àmbito (*scope*): *istanza*, se l'attributo appartiene al singolo oggetto, *statico* se appartiene alla classe, cioè è condiviso fra tutte le istanze della classe, analogamente ai campi statici del C++ o del Java.

molteplicità: índica se l'attributo può essere replicato, cioè avere piú valori (si può usare per rappresentare un array).

valore iniziale: valore assegnato all'attributo quando si crea l'oggetto.

La sintassi UML per gli attributi è la seguente:

```
<visibilita‘> <nome> <molteplicita‘> : <tipo> = <val-iniziale>
```

Se un attributo ha scope statico, viene sottolineato.

La visibilità si rappresenta con i seguenti simboli: '`+`' (pubblica), '`#`' (protetta), '`~`' (package) e '`-`' (privata).

La molteplicità si índica con un numero o un intervallo numerico fra parentesi quadre, come nei seguenti esempi:

`[3]`	tre valori
`[1..4]`	da uno a quattro valori;
`[1..*]`	uno o piú valori;
`[0..1]`	zero o un valore (attributo opzionale).

Operazioni

Ogni operazione viene identificata da una *segnatura* (*signature*) costituita dal nome della funzione e dalla *lista dei parametri*, eventualmente vuota. Per ciascun parametro si specificano il nome e, opzionalmente, le seguenti informazioni:

direzione: *ingresso* (`in`), *uscita* (`out`), *ingresso e uscita* (`inout`).

tipo del parametro.

valore default: valore passato al metodo che implementa la funzione, se l'argomento corrispondente al parametro non viene specificato.

Inoltre le operazioni, analogamente agli attributi, hanno scope, visibilità e tipo; quest'ultimo è il tipo dell'eventuale valore restituito. Anche queste informazioni sono opzionali.

La sintassi per le operazioni è la seguente:

```
<visibilita‘> <nome> (<lista-parametri>) : <tipo>
```

dove ciascun parametro della lista ha questa forma:

```
<direzione> <nome> : <tipo> = <val-default>
```

Solo il nome del parametro è obbligatorio, ed i parametri sono separati da virgole.

È utile osservare la distinzione fra *operazione* e *metodo*. Un'operazione è la specifica di un comportamento, specifica che si può ridurre alla semplice descrizione dei parametri o includere vincoli (per esempio, precondizioni, invarianti e postcondizioni) e varie annotazioni. Un metodo è l'implementazione di un'operazione. L'UML non ha elementi di modello destinati a descrivere i metodi, che comunque non interessano in fase di specifica. Se bisogna descrivere l'implementazione di un'operazione, per esempio nella fase di codifica, si possono usare delle note associate all'operazione o altri metodi offerti dallo strumento CASE.

Oggetti

Un oggetto viene rappresentato da un rettangolo contenente i nomi dell'oggetto e della classe d'appartenenza, sottolineati e separati dal carattere ':', ed opzionalmente gli attributi con i rispettivi valori. Il nome dell'oggetto può mancare, e in questo caso il nome della classe viene preceduto dal carattere ':'. È possibile esprimere degli stereotipi come nella rappresentazione delle classi.

Nei modelli UML la rappresentazione esplicita di oggetti è piuttosto rara, in quanto il lavoro di analisi, specifica e progettazione si basa essenzialmente sulle classi e le loro relazioni, però gli oggetti possono essere usati per esemplificare delle situazioni particolari o tipiche. La figura 4.9 mostra la rappresentazione di una classe e di una sua istanza.

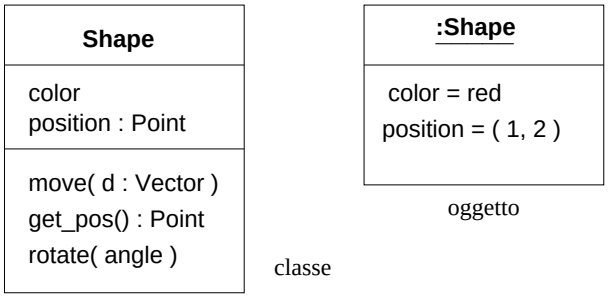


Figura 4.9: Una classe ed un oggetto.

4.6.3 Associazioni e link

In UML sono possibili associazioni di qualsiasi arietà (binarie, ternarie, ...), ma tratteremo solo quelle binarie.

Un'associazione binaria è definita dai suoi *estremi* (*end*), ciascuno dei quali comprende

- *nome della classe* corrispondente all'estremo;
- *ruolo* della classe nell'associazione;
- *molteplicità*;
- altre proprietà, quali l'*ordinamento* e l'*unicità*.

Un'associazione può avere un nome.

Il ruolo di un estremo permette di descrivere sinteticamente la relazione fra le istanze della classe corrispondente rispetto a quelle della classe situata all'estremo opposto.

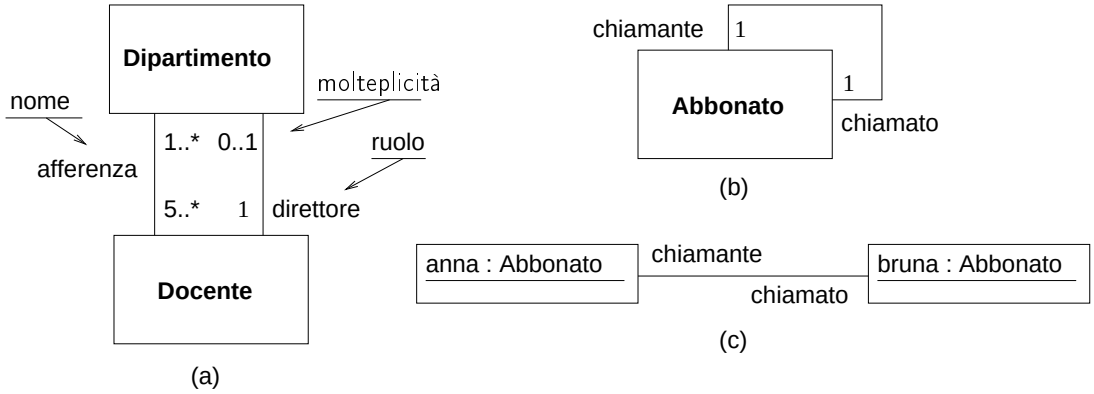


Figura 4.10: Associazioni: molteplicità e ruoli

La molteplicità di una classe in un'associazione è il numero di istanze di tale classe che possono essere in relazione con una istanza della classe associata. La molteplicità può essere indicata con intervalli numerici alle estremità della linea che rappresenta l'associazione: per esempio, `1`, `0..1` (zero o uno), `1..*` (uno o piú), `0..*` (zero o piú), `*` (equivalente a `0..*`). Se la molteplicità non è indicata esplicitamente nel diagramma, si possono avere questi casi: (i) si è scelto di elidere la molteplicità dalla presentazione grafica, oppure (ii) la molteplicità è *zero o piú*, oppure (iii) il modello è incompleto. È utile informarsi su come lo strumento CASE usato permette di gestire la presentazione grafica della molteplicità.

L'ordinamento e l'unicità specificano rispettivamente, quando la molteplicità è maggiore di uno, se le istanze di una classe associate ad una istanza dell'altra sono ordinate o no, e se alcune istanze possono avere duplicati[15].

La fig. 4.10(a) mostra due associazioni. L'associazione di nome "afferenza" fornisce queste informazioni:

- un docente può afferire ad *uno o piú* dipartimenti (molteplicità `1 .. *` all'estremo **Dipartimento**);
- ad un dipartimento afferiscono *almeno cinque* docenti (molteplicità `5 .. *` all'estremo **Docente**).

L'associazione anonima fra **Dipartimento** e **Docente** fornisce queste informazioni:

- un docente può partecipare all'associazione con *uno o nessun* dipartimento (molteplicità `0 .. 1` all'estremo **Dipartimento**);

[15]La duplicazione è possibile per le istanze di tipi di dati astratti.

- ad un dipartimento è associato *esattamente un* docente col ruolo "direttore" (molteplicità 1 all'estremo **Docente**).

Osservazione. Il diagramma non specifica se il direttore deve afferire al dipartimento!

La fig. 4.10(b) mostra un'associazione che rappresenta legami *fra istanze di una stessa classe*, in cui esattamente un abbonato ha il ruolo di chiamante ed esattamente un abbonato ha il ruolo di chiamato. La fig. 4.10(c) mostra un'istanza (link) dell'associazione.

Un'altra informazione applicabile ad un'estremità di associazione è il *qualificatore*, che distingue fra di loro i diversi oggetti di una classe che possono essere in relazione con oggetti dell'altra. Il qualificatore rappresenta cioè un'informazione (p.es. un codice identificativo) che permette di individuare una particolare istanza, fra molte, di una classe associata. Nella fig. 4.11, la penultima associazione mostra che un utente può avere piú account, e nell'associazione successiva questa molteplicità viene ridotta a uno, grazie al qualificatore uid che identifica i singoli account.

Il qualificatore viene rappresentato da un rettangolo avente un lato combaciante con un lato della classe opposta a quella di cui si vogliono qualificare le istanze: nella fig. 4.11 il qualificatore uid serve a qualificare cioè selezionare le istanze di **Account**.

Altre proprietà, il *tipo di aggregazione* (*aggregation kind*) e la navigabilità, verranno descritte piú oltre.

Un'associazione (o un link), nella forma piú semplice, si rappresenta come una linea, eventualmente spezzata o curva, fra le classi (o oggetti) coinvolte. Se un'associazione coinvolge piú di due classi, si rappresenta con una losanga unita da linee alle classi coinvolte (analogamente per i link).

Alcuni semplici modi di rappresentare le associazioni sono mostrati in fig. 4.11: la classe **User** rappresenta gli utenti di un sistema di calcolo, e la classe **Account** rappresenta i relativi account, ciascuno contraddistinto da un numero identificatore (uid).

Un'associazione può avere un nome descrittivo, per esempio afferenza in fig. 4.10(a).

Aggregazione

La *aggregazione* è un'associazione che lega un'entità complessa (aggregato) alle proprie parti componenti. In UML, un'associazione è un'aggregazione se la proprietà *tipo di aggregazione* all'estremità relativa all'entità complessa ha il valore *shared*. Nei diagrammi di classi e di oggetti, una relazione di aggregazione viene indicata da una piccola losanga vuota all'estremità dell'associazione dalla parte della classe (o dell'oggetto) che rappresenta l'entità complessa.

La differenza fra un'associazione pura e semplice e un'aggregazione non è netta. L'aggregazione è un'annotazione aggiuntiva che esprime il concetto di "appartenenza", "contenimento", "ripartizione", o in generale di una forma di subordinazione strutturale non

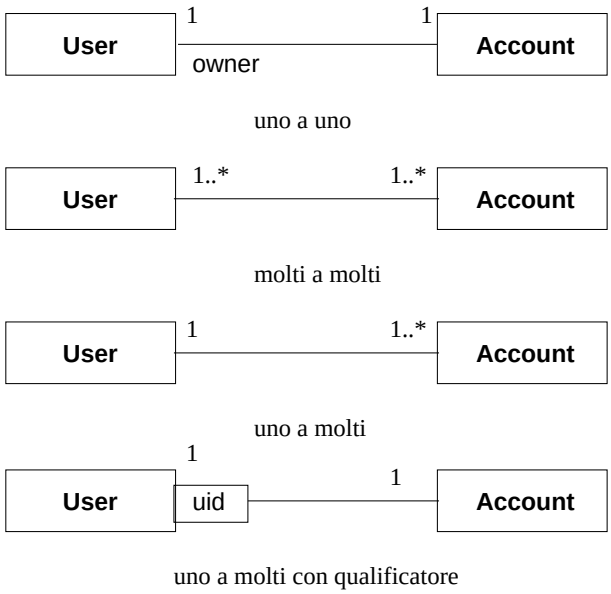


Figura 4.11: Associazioni.

rigida. Per esempio, nella fig. 4.12 la relazione fra una banca e i suoi clienti viene modellata da una semplice associazione, poiché i clienti non "fanno parte" della loro banca, mentre la relazione fra una squadra e i suoi giocatori si può modellare piú accuratamente con una aggregazione. In questo secondo caso, però, sarebbe stata accettabile anche una semplice associazione.

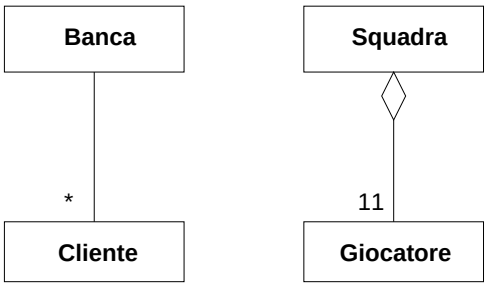


Figura 4.12: Associazioni e aggregazioni.

Un'istanza di una classe può appartenere a piú istanze di aggregati. La fig. 4.13 mostra che la classe **Studente** partecipa, come componente, alle aggregazioni con le classi **Squadra** e **Coro**. Il diagramma di oggetti nella stessa figura mostra una possibile configurazione di istanze compatibile col diagramma delle classi: la studentessa anna appartiene a due istanze della classe **Squadra**, lo studente beppe appartiene ad un'istanza della classe **Squadra** e ad una della classe **Coro**, lo studente carlo appartiene a un'istanza della classe **Coro**.

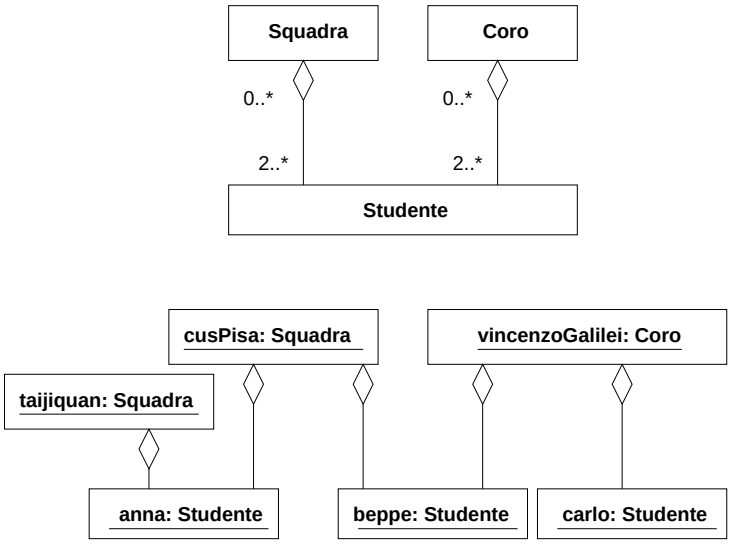


Figura 4.13: Aggregazioni multiple.

4.6.4 Composizione

La *composizione* modella una subordinazione strutturale rigida, tale cioè che l'aggregato abbia il completo e unico controllo delle parti componenti. Mentre nell'aggregazione le parti sono indipendenti dall'aggregato (esistono anche al di fuori dell'aggregazione), nella composizione esiste una dipendenza stretta fra composto e componenti. Spesso la composizione rappresenta una situazione in cui l'esistenza dei componenti coincide con quella del composto, per cui la creazione e la distruzione del composto implicano la creazione e la distruzione dei componenti. In ogni caso, un componente può appartenere ad un solo composto, e il composto è il "padrone" del componente.

In un modello, un'associazione è una composizione se la proprietà *tipo di aggregazione* all'estremità relativa all'entità complessa ha il valore *composite*. La composizione si rappresenta con una losanga nera dalla parte dell'entità complessa, oppure si possono disegnare i componenti all'interno dell'entità stessa. La fig. 4.14 mostra queste due notazioni.

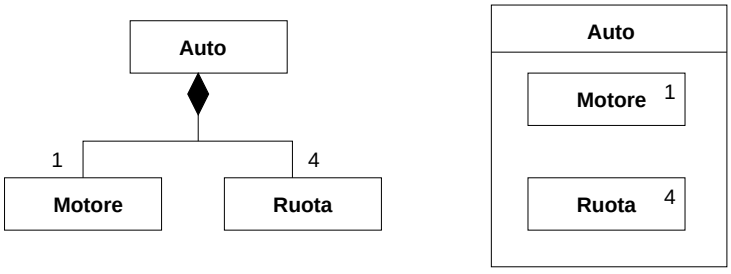


Figura 4.14: Composizione.

Una classe può appartenere come componente a piú di una composizione, ma un'istanza di una classe componente può appartenere ad una sola istanza di una classe composta.

Nella fig. 4.15 la classe **Motore** è in relazione di composizione con **Nave** e **Auto**, ma una qualsiasi sua istanza può essere componente di una sola istanza di una delle due classi.

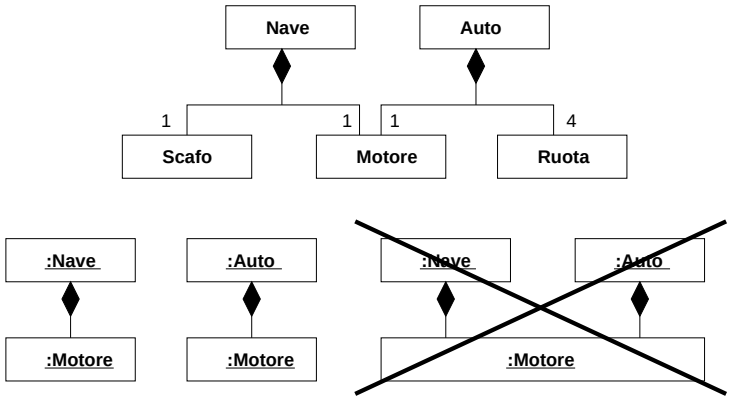


Figura 4.15: Composizione fra classi e fra oggetti.

Infine, la fig. 4.16 mostra un semplice diagramma delle classi con associazioni, aggregazioni e composizioni.

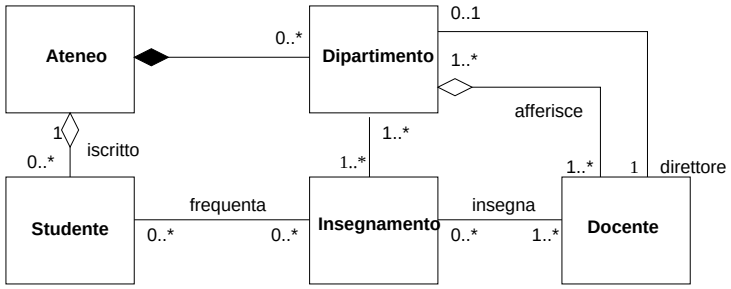


Figura 4.16: Un diagramma delle classi.

4.6.5 Generalizzazione

Una classe (che chiameremo *classe base* o *superclasse*) *generalizza* un'altra classe (che chiameremo *classe derivata* o *sottoclasse*) quando definisce un insieme di elementi che include l'insieme di elementi definiti dalla classe derivata: quest'ultima, cioè, rappresenta un sottoinsieme della classe base.

Una classe base può avere piú classi derivate, e in questo caso ne riassume alcune caratteristiche comuni (attributi, operazioni, associazioni, vincoli...). Le caratteristiche di tale classe, cioè, definiscono un insieme di oggetti che include l'unione degli insiemi di oggetti definiti dalle sottoclassi. Un oggetto appartenente ad una classe derivata, quindi, appartiene anche alla classe base. È possibile che ogni oggetto appartenente alla classe base appartenga ad almeno una delle classi derivate, oppure che alcuni oggetti appartengano solo alla classe base.

La relazione di generalizzazione si può anche chiamare *specializzazione*, cambiando il punto di vista ma restando immutato il significato: una classe è una specializzazione di un'altra classe se aggiunge delle caratteristiche alla sua struttura o al suo comportamento (è quindi un'*estensione*) oppure vi aggiunge dei vincoli (si ha quindi una *restrizione*).

Per esempio, data una classe **Rectangle** (fig. 4.17) con gli attributi center, width e height, la classe Colored è un'estensione, poiché aggiunge l'attributo colore, mentre la classe **Square** è una restrizione, poiché pone il vincolo che gli attributi width e height abbiano lo stesso valore. Osserviamo, a scanso di equivoci, che anche un'estensione è un sottoinsieme della superclasse.

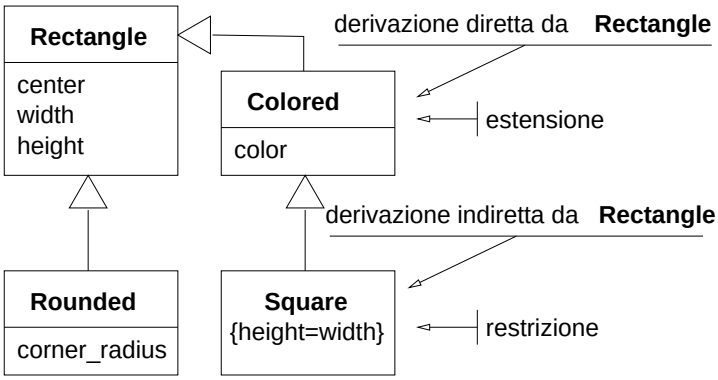


Figura 4.17: Generalizzazione.

Il fatto che le istanze di una classe derivata siano un sottoinsieme delle istanze della classe base si esprime anche col *principio di sostituzione* (di Barbara Liskov): un'istanza della classe derivata può sostituire un'istanza della classe base. Questo principio è un utile criterio per valutare se una certa classe può essere descritta come una specializzazione di un'altra. È particolarmente importante verificare l'applicabilità del principio della Liskov nel caso in cui la classe derivata venga ottenuta per restrizione dalla classe base, cioè aggiungendo dei vincoli.

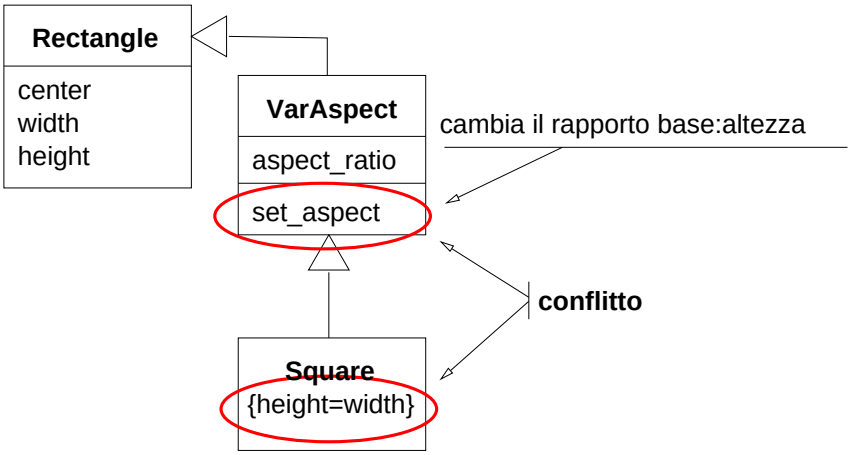


Figura 4.18: Generalizzazione e principio di sostituzione.

Supponiamo, per esempio, che una classe **VarAspect** (fig. 4.18) derivata da **Rectangle** abbia un'operazione set_aspect che cambia il rapporto base/altezza (lasciando invariata, poniamo, la lunghezza della base). Se si definisce una classe **Square** come restrizione di **VarAspect**, cioè introducendo il vincolo che base e altezza abbiano lo stesso valore, si víola il principio di sostituzione, poiché l'operazione set_aspect non è applicabile alle istanze di **Square**.

Poiché una sottoclasse ha tutte le caratteristiche della superclasse, si dice che tali caratteristiche vengono *ereditate*. In particolare, vengono ereditate le operazioni, di cui si può effettuare anche una *ridefinizione* (*overriding*) nella classe derivata. In una classe derivata, cioè, si può avere un'operazione che ha la stessa *segnatura* (nome dell'operazione, tipo del valore restituito, numero, tipo e ordine degli argomenti) di un'operazione della classe base, ma una diversa implementazione. Naturalmente la ridefinizione è utile se la nuova implementazione è compatibile col significato originario dell'operazione, altrimenti non varrebbe il principio di sostituzione. Per esempio, consideriamo una classe **Studente** con l'operazione iscrivi(), che rappresenta l'iscrizione dello studente secondo la procedura normale. Se una categoria di studenti, per esempio quelli già in possesso di una laurea, richiede una procedura diversa, si può definire una classe **StudenteLaureato**, derivata da **Studente**, che ridefinisce opportunamente l'operazione iscrivi().

Una classe derivata può essere ulteriormente specializzata in una o piú classi, e una classe base può essere ulteriormente generalizzata (finché non si arriva alla classe universale che comprende tutti i possibili oggetti). Quando si hanno delle catene di generalizzazioni, può essere utile distinguere le classi basi o derivate *dirette* da quelle *indirette*, a seconda che si ottengano "in un solo passo" o no, a partire da un'altra classe. Le classi base e derivate indirette si chiamano anche, rispettivamente, *antenati* e *discendenti*.

La generalizzazione si rappresenta con una freccia terminante in un triangolo vuoto col vertice che tocca la superclasse.

Classi astratte e concrete

Una classe è *concreta* se esistono degli oggetti che siano istanze *dirette* di tale classe, cioè appartengano ad essa e non a una classe derivata. Se consideriamo due o piú classi concrete che hanno alcune caratteristiche in comune, possiamo generalizzarle definendo una classe base che riassume queste caratteristiche. Può quindi accadere che non possano esistere delle istanze dirette di questa classe base, che si dice allora *astratta*.

Per esempio (fig. 4.19), consideriamo delle classi come **Uomo**, **Lupo**, **Megattera** eccetera[16]. Gli animali appartenenti a queste classi hanno delle caratteristiche in comune (per esempio, sono omeotermi e vivipari), che si possono riassumere nella definizione di una classe base *Mammifero*. Questa classe è astratta perché non esiste un animale che sia un mammifero senza appartenere anche a una delle classi derivate.

[16]Ovviamente stiamo usando il termine "classe" in modo diverso da come viene usato nelle scienze naturali: *Homo sapiens* è una specie, appartenente alla classe *Mammalia*.

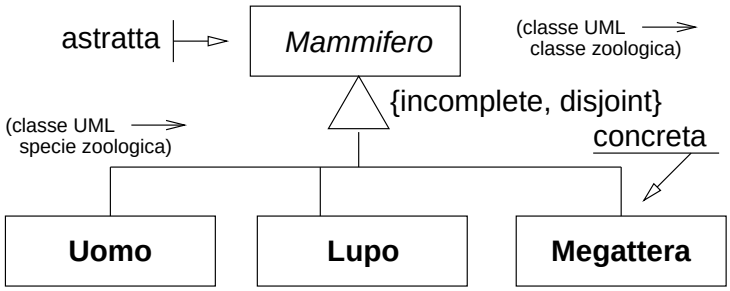


Figura 4.19: Esempio di classi astratte e concrete.

Il significato delle proprietà incomplete e disjoint mostrate nella figura verrà spiegato piú oltre.

In UML l'astrattezza di una classe (e di altri elementi di modello) si specifica con la proprietà abstract. Per convenzione, nei diagrammi i nomi delle entità astratte si scrivono in caratteri obliqui; se non è pratico scrivere in obliquo (per esempio in un diagramma fatto a mano) si può scrivere la parola abstract fra parentesi graffe sotto al nome della classe.

Eredità multipla

Si parla di *eredità multipla* quando una classe derivata è un sottoinsieme di due o piú classi che non sono in relazione di generalizzazione o specializzazione fra di loro. In questo caso, le istanze della classe derivata ereditano le caratteristiche di tutte le classi base.

Insiemi di generalizzazioni

Nei modelli di analisi la relazione di generalizzazione viene usata spesso per classificare le entità del dominio analizzato, mettendone in evidenza le reciproche affinità e differenze. Per esempio, può essere utile classificare i prodotti di un'azienda, i suoi clienti o (fig. 4.22) i suoi dipendenti.

Per descrivere precisamente una classificazione, l'UML mette a disposizione il concetto di *insieme di generalizzazioni*. Informalmente, un insieme di generalizzazioni è un modo di raggruppare le sottoclassi di una classe base, cioè un insieme di sottoinsiemi, a cui si può dare un nome che descriva il criterio con cui si raggruppano le sottoclassi. Un insieme di generalizzazioni è *completo* se le sue classi comprendono tutte le categorie previste dal criterio di classificazione rappresentato da tale insieme, e *disgiunto* se le sottoclassi sono disgiunte (l'intersezione di ciascuna coppia di sottoclassi è vuota). Per default, un insieme di generalizzazioni è incompleto e disgiunto.

Nell'esempio di fig. 4.20 si suppone che una persona possa fare un solo lavoro; l'insieme di generalizzazioni è quindi disgiunto, ed è incompleto perché evidentemente esistono molte altre categorie di lavoratori. In fig. 4.21, invece, si suppone che una persona

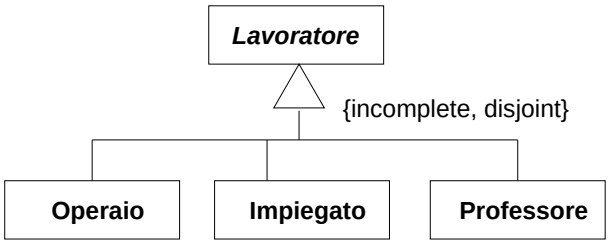


Figura 4.20: Insieme di generalizzazioni incompleto e disgiunto.

possa praticare piú di uno sport, quindi l'insieme di generalizzazioni è overlapping (non disgiunto).

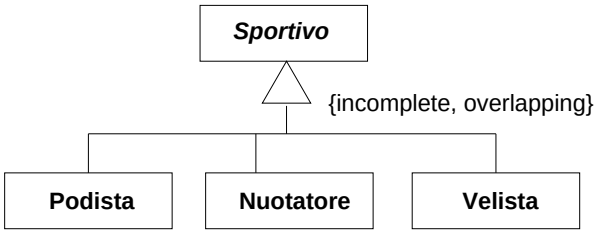


Figura 4.21: Insieme di generalizzazioni incompleto e non disgiunto.

Nell'esempio di fig. 4.22 l'insieme dei dipendenti di un'azienda viene classificato secondo due criteri ortogonali, retribuzione e mansione. Si hanno quindi due insiemi di generalizzazioni, ognuno completo e disgiunto. Ogni dipendente è un'istanza sia di una classe di retribuzione (**Livello1** eccetera) che di una classe di mansioni (**Tecnico** o **Amministrativo**). Poiché nessun dipendente può appartenere *solo* a una classe di retribuzione o a una classe di mansioni, tutte le classi della fig. 4.22 (a) sono astratte. Nella fig. 4.22 (b) si mostra una classe concreta costruita per eredità multipla da una classe di retribuzione e una classe di mansioni.

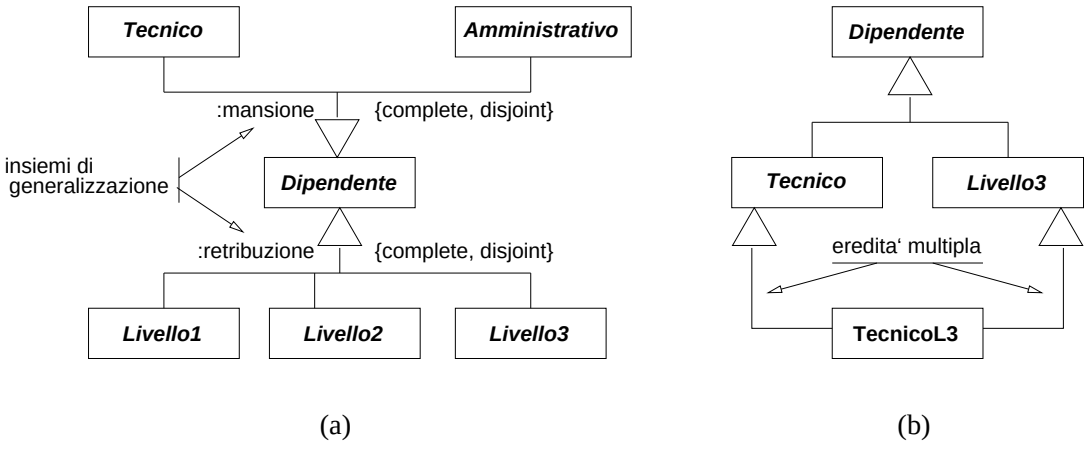


Figura 4.22: Insiemi di generalizzazione.

Osservazione. L'esempio mostrato in fig. 4.22 serve a chiarire il significato degli insiemi di generalizzazioni, ma non è necessariamente il modo migliore per modellare la situazione

presa ad esempio. Si può osservare, infatti, che un dipendente rappresentato da un'istanza di **TecnicoL3**, secondo questo modello, non può cambiare mansione né classe di retribuzione, poiché la relazione di generalizzazione è statica e fissa rigidamente l'appartenenza delle istanze di **TecnicoL3** alle superclassi *Tecnico* e *Livello3*. Si avrebbe un modello piú realistico considerando la retribuzione e la mansione come concetti *associati* ai dipendenti, come mostra la fig. 4.23. I link fra istanze possono essere creati e distrutti dinamicamente, per cui le associazioni permettono di costruire strutture logiche piú flessibili rispetto alla generalizzazione.

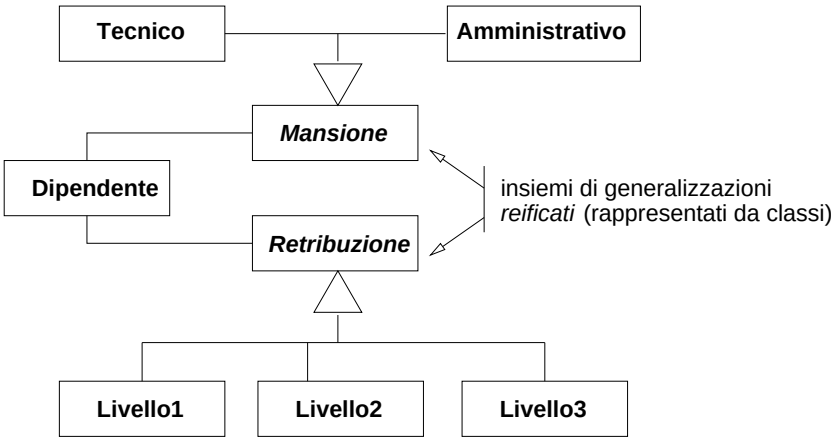


Figura 4.23: Associazioni invece di generalizzazioni.

4.6.6 Diagrammi dei casi d'uso

Un diagramma dei casi d'uso schematizza il comportamento del sistema dal punto di vista degli utenti, o, piú in generale, di altri sistemi che interagiscono col sistema specificato. Un *attore* rappresenta un'entità esterna al sistema, un *caso d'uso* rappresenta un servizio offerto dal sistema. Ciascun servizio viene espletato attraverso sequenze di messaggi scambiati fra gli attori ed il sistema.

Attori e casi d'uso sono legati da associazioni che rappresentano comunicazioni. I casi d'uso *non rappresentano sottosistemi*, per cui non possono interagire, cioè scambiarsi messaggi, fra di loro, ma possono essere legati da relazioni di *inclusione*, *estensione* e *generalizzazione*.

Il comportamento richiesto al sistema per fornire il servizio rappresentato da un caso d'uso può essere specificato in vari modi, per esempio per mezzo di macchine a stati o di descrizioni testuali. In fase di analisi può essere sufficiente una descrizione in linguaggio naturale, oppure una descrizione piú formale delle di sequenze di interazioni (*scenari*) fra attori e sistema previste per lo svolgimento del servizio.

La fig. 4.24 mostra un semplice diagramma di casi d'uso, relativo a un sistema di pagamento POS (*Point Of Sale*). Gli attori sono il cassiere e il cliente, i servizi forniti dal sistema sono il pagamento, il rimborso e il login; quest'ultimo coinvolge solo il cassiere. Il

servizio di pagamento ha due possibili estensioni, cioè comportamenti aggiuntivi rispetto a quello del caso d'uso fondamentale.

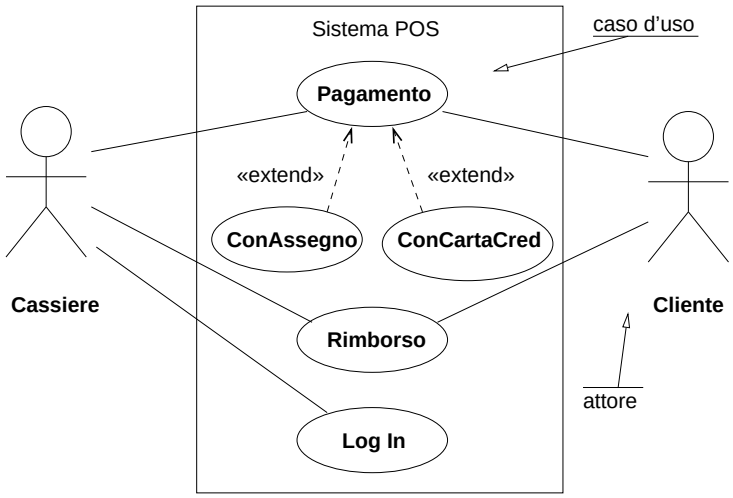


Figura 4.24: Una diagramma di casi d'uso.

4.6.7 Diagrammi di stato

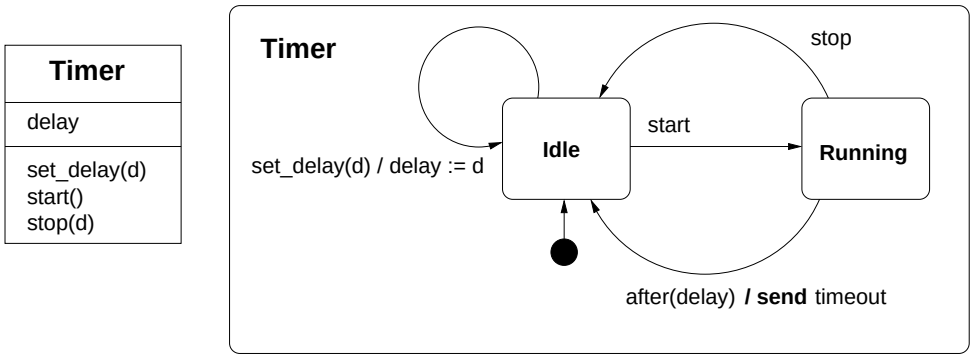


Figura 4.25: Una macchina a stati associata ad una classe.

Il *diagramma di stato* è uno dei diagrammi che fanno parte del modello dinamico di un sistema, e rappresenta una *macchina a stati* associata a una classe (fig. 4.25), ad una *collaborazione* (interazione di un insieme di oggetti), o ad un'operazione. La macchina a stati descrive l'evoluzione temporale degli oggetti e la loro risposta agli stimoli esterni.

Le macchine a stati impiegate nell'UML sono basate sul formalismo degli automi a stati finiti visto in precedenza, che qui, però, viene esteso notevolmente in modo da renderlo piú espressivo e sintetico. Nell'UML viene usata la notazione degli *Statechart* [20], che permette una descrizione strutturata degli stati di un automa. Questa notazione è ricca di concetti e di varianti grafiche, e ne saranno illustrati solo gli aspetti principali.

I concetti fondamentali rappresentati dal diagramma di stato sono:

occorrenze: avvenimenti associati a istanti nel tempo e (spesso implicitamente) a punti nello spazio. Per esempio, se viene premuto un tasto di un terminale, questa è un'occorrenza della pressione di un tasto. Quando lo stesso tasto (o un altro) viene premuto di nuovo, è un'altra occorrenza.

eventi: insiemi di occorrenze di uno stesso tipo. Per esempio, l'evento **TastoPremuto** potrebbe rappresentare l'insieme di tutte le possibili occorrenze della pressione di un tasto. Nel séguito, però, spesso si userà il termine "evento" al posto di "occorrenza".

stati: situazioni in cui un oggetto soddisfa certe condizioni, esegue delle attività, o semplicemente aspetta degli eventi.

transizioni: conseguentemente al verificarsi di un evento, un oggetto può modificare il proprio stato: una transizione è il passaggio da uno stato ad un altro e non è interrompibile, per cui si considera *informalmente* come istantaneo.

azioni: possono essere eseguite in corrispondenza di transizioni, e non sono interrompibili, per cui si possono interpretare come se fossero istantanee.

attività: possono essere associate agli stati, possono essere interrotte dal verificarsi di eventi che causano l'uscita dallo stato, ed hanno una durata non nulla.

Eventi

Un evento può essere una (*ricezione di*) *chiamata di operazione*, un *cambiamento*, una *ricezione di segnale*, un *evento temporale*, o un *evento di completamento*.

Un evento di chiamata avviene quando viene ricevuta una richiesta di esecuzione di un'operazione. Osserviamo che normalmente si modellano come eventi solo quelle chiamate che causano un cambiamento di stato nell'oggetto che le riceve. Per esempio, la chiamata di un'operazione che si limita a restituire un valore oppure a cambiare un attributo, ma non il comportamento di un oggetto, normalmente non viene modellata come un evento.

Un evento di cambiamento è il passaggio del valore di una condizione logica da falso a vero, e si rappresenta con la parola **when** seguita da un'espressione booleana. Per esempio, l'espressione **when** (t $\geq$ t_max) in fig. 4.26 rappresenta l'evento "*t raggiunge o supera la soglia t_max*".

I segnali sono messaggi che gli oggetti si possono scambiare per comunicare fra di loro, e possono essere strutturati in una gerarchia di generalizzazione: per esempio, un ipotetico segnale **Input** può essere descritto come generalizzazione dei segnali **Mouse** e **Keyboard**, che a loro volta possono essere ulteriormente specializzati. Una gerarchia di segnali si rappresenta graficamente in modo simile ad una gerarchia di classi. Un segnale si rappresenta come un rettangolo contenente lo stereotipo ≪signal≫, il nome del segnale ed eventuali attributi.

Un evento temporale si verifica quando il tempo assume un particolare valore assoluto (per esempio, il 31 dicembre 1999), oppure quando è trascorso un certo periodo dall'istante in cui il sistema è entrato in un certo stato (per esempio, dieci secondi dopo l'apertura di una valvola). Gli eventi temporali del primo tipo si rappresentano con la parola **at** seguita

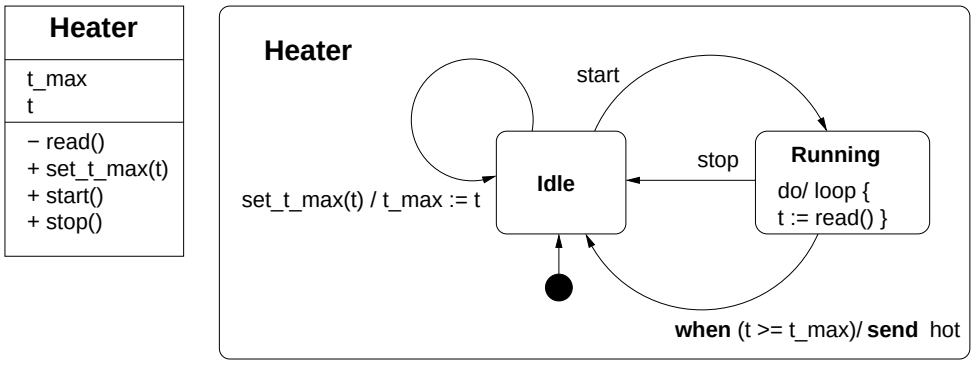


Figura 4.26: Eventi di cambiamento.

da una condizione booleana (per esempio, **at** (date = 2002-12-31)), quelli del secondo tipo con la parola **after** seguita da una durata (per esempio, **after** (10 ms)).

Un evento di completamento si verifica quando termina una attività associata ad uno stato.

Gestione degli eventi

Se un evento si verifica nel corso di una transizione, non ha influenza sull'eventuale azione associata alla transizione (ricordiamo che le azioni non sono interrompibili) e viene accantonato in una riserva di eventi (*event pool*) per essere considerato nello stato successivo. Se in quest'ultimo stato l'evento non abilita alcuna transizione, viene cancellato.

Se, mentre un oggetto si trova in un certo stato, si verifica un evento che non innesca transizioni uscenti da quello stato, l'oggetto si può comportare in due modi: (i) questo evento viene cancellato e quindi non potrà piú influenzare l'oggetto (è come se non fosse mai accaduto), oppure (ii) se nello stato in questione l'evento è stato dichiarato come *differibile*, viene tenuto da parte finché l'oggetto non entra in uno stato in cui tale evento non è piú dichiarato come differibile. In questo nuovo stato, l'evento cosí memorizzato o innesca una transizione, o viene perduto definitivamente. Gli eventi differibili in uno stato vengono dichiarati come tali nel simbolo dello stato, con la parola defer.

Stati

Gli stati si rappresentano come rettangoli ovalizzati contenenti opzionalmente il nome dello stato[17], un'eventuale attività (preceduta dalla parola do) ed altre informazioni che vedremo piú oltre. In particolare, uno stato può contenere dei sottostati. Il simbolo di uno stato può contenere la parola entry seguita dal nome di un'azione (separato da una barra) (fig. 4.27). Questo significa che l'azione deve essere eseguita ogni volta che l'oggetto entra nello stato in questione. Analogamente, la parola exit etichetta un'azione da eseguire all'uscita dallo stato. Una coppia *evento*/*azione* entro il simbolo di uno stato

[17]Si possono avere stati anonimi.

significa che al verificarsi dell'evento viene eseguita l'azione corrispondente, e l'oggetto resta nello stato corrente (*transizione interna*). In questo caso non vengono eseguite le eventuali azioni di entry o di exit.

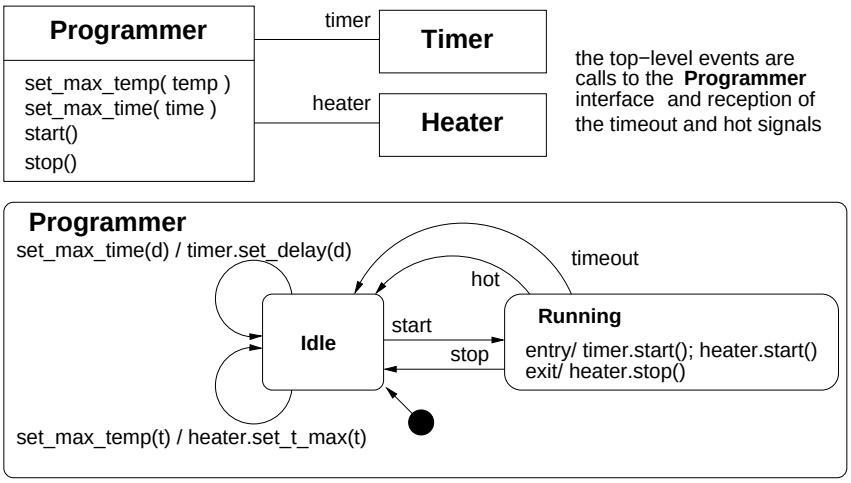


Figura 4.27: Azioni di entry ed exit.

Per indicare lo stato iniziale[18] si usa una freccia che parte da un cerchietto nero e raggiunge lo stato iniziale. Uno stato finale viene rappresentato da un cerchio contenente un cerchietto annerito (un "bersaglio"). Una macchina a stati può non avere uno stato finale, per modellare un'attività teoricamente perpetua, che negli automi a stati finiti ímplica un comportamento ciclico.

Transizioni

Le transizioni sono rappresentate da frecce fra gli stati corrispondenti, etichettate col nome dell'evento causante la transizione, con eventuali attributi dell'evento, con una condizione (*guardia*) necessaria per l'abilitazione della transizione (racchiusa fra parentesi quadre), e con un'azione da eseguire, separata dalle informazioni precedenti per mezzo di una barra obliqua. Ciascuna di queste tre informazioni è opzionale. Se manca l'indicazione dell'evento, la transizione avviene al termine dell'attività associata allo stato di partenza: si tratta di una transizione *di completamento*.

Le guardie sono espressioni logiche che possono essere scritte nella normale notazione matematica o in OCL (*Object Constraint Language*), un linguaggio dichiarativo con una notazione specifica per riferirsi ad elementi di modelli UML. La valutazione delle guardie non può avere effetti collaterali.

Un'azione può inviare dei segnali, e in questo caso si usa la parola **send** seguita dal nome e da eventuali parametri del segnale. L'invio di un segnale si può rappresentare

[18] Un automa deterministico ha uno ed un solo stato iniziale, mentre un automa non deterministico può averne zero o piú.

anche graficamente, mediante una figura a forma di cartello indicatore (un rettangolo con una "punta" triangolare), etichettata col nome e i parametri del segnale.

Macchine a stati gerarchiche

La descrizione del modello dinamico generalmente è gerarchica, cioè articolata su diversi livelli di astrazione, in ciascuno dei quali alcuni elementi del livello superiore vengono raffinati ed analizzati.

Un'attività associata ad uno stato può quindi essere descritta a sua volta da una macchina a stati. Questa avrà uno stato iniziale ed uno o piú stati finali. Il sottodiagramma che descrive l'attività può sempre essere disegnato entro il simbolo dello stato che la contiene. Se non ci sono transizioni che attraversano il confine del sottodiagramma, questo può essere disegnato separatamente.

In generale, qualsiasi stato (*superstato*) può essere decomposto in *sottostati*, che ereditano le transizioni che coinvolgono il superstato. I sottostati possono essere *sequenziali*, se un solo stato alla volta è attivo, o *concorrenti*, se piú stati sono attivi contemporaneamente.

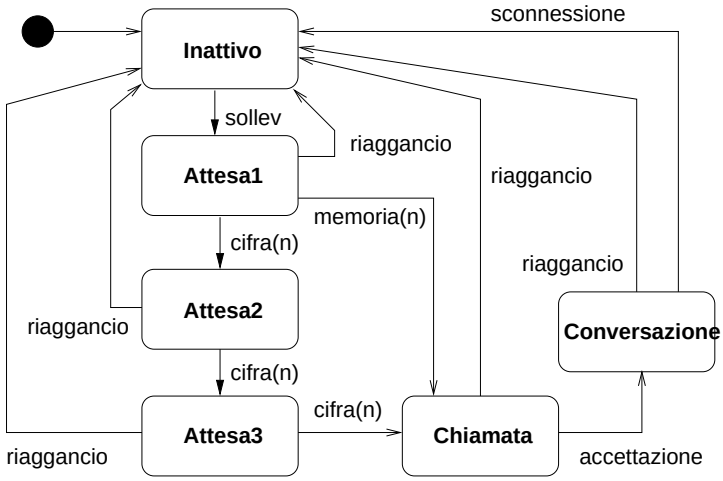


Figura 4.28: Una macchina a stati non gerarchica.

Consideriamo, per esempio, la macchina a stati associata all'interazione fra l'utente e un centralino (fig. 4.28). Si suppone che l'utente possa comporre numeri di tre cifre, oppure premere un tasto che seleziona un numero memorizzato. Il diagramma di questa macchina a stati non sfrutta la possibilità di composizione gerarchica offerta dagli Statechart, per cui le transizioni causate dagli eventi riaggancio devono essere mostrate per ciascuno stato successivo a quello iniziale.

Il diagramma si semplifica se raggruppiamo questi stati in un superstato (**Attivo**) composto da sottostati sequenziali e ridisegnamo le transizioni come in fig. 4.29. La transizione in ingresso al superstato **Attivo** porta la macchina nel sottostato iniziale (**Attesa1**), mentre la transizione di completamento fra i due stati ad alto livello avviene quando la

sottomacchina dello stato **Attivo** termina il proprio funzionamento. La transizione attivata dagli eventi riaggancio viene ereditata dai sottostati: questo significa che, in qualsiasi sottostato di **Attivo**, il riaggancio riporta la macchina nello stato **Inattivo**.

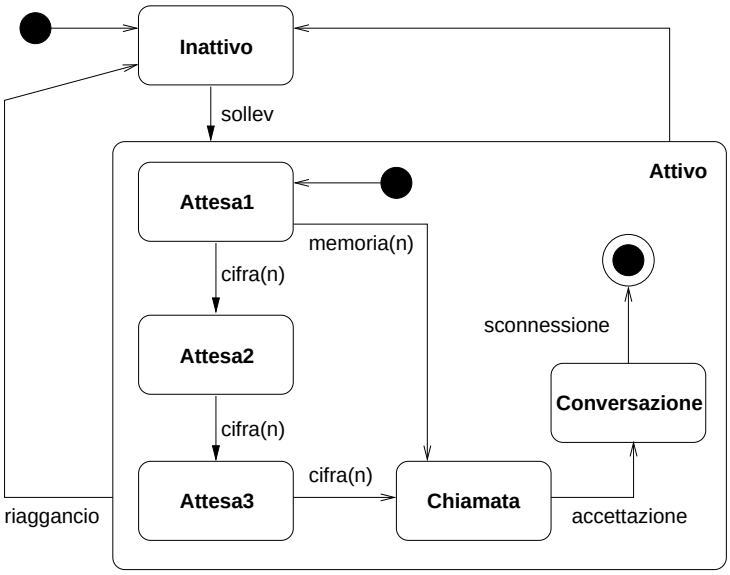


Figura 4.29: Una macchina a stati gerarchica.

Stati concorrenti

Uno stato può essere scomposto anche in *regioni concorrenti*, che descrivono attività concorrenti nell'àmbito dello stato che le contiene. Queste attività, a loro volta, sono descritte da macchine a stati. La fig. 4.30 mostra il funzionamento di un termoventilatore, in cui il controllo della velocità e quello della temperatura sono indipendenti.

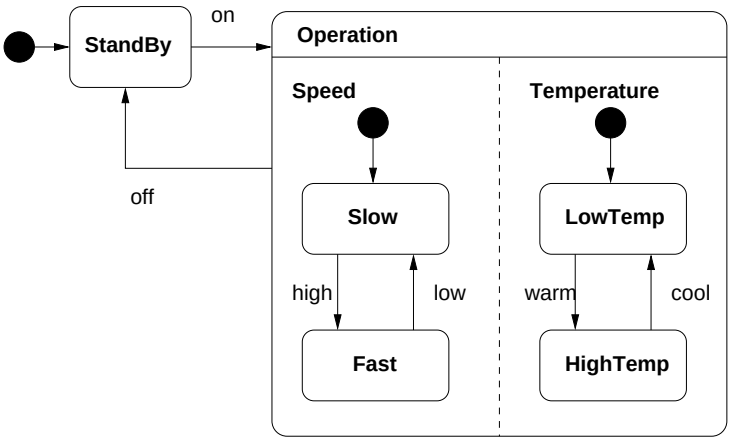


Figura 4.30: Una macchina a stati gerarchica con regioni concorrenti.

Gli stati concorrenti possono interagire attraverso *eventi condivisi*, *scambio di segnali*, *parametri* degli eventi o dei segnali, e *attributi* dell'oggetto a cui appartiene la macchina

a stati. Le interazioni possono avvenire anche attraverso la valutazione delle espressioni che costituiscono le guardie e le azioni associate alle transizioni.

L'esecuzione delle azioni può modificare gli attributi condivisi fra stati concorrenti, però è bene evitare, finché possibile, di modellare in questo modo l'interazione fra stati concorrenti. Questo meccanismo di interazione, infatti, è poco strutturato e poco leggibile, e rende piú probabili gli errori nella specifica o nella realizzazione del sistema. La valutazione delle guardie, invece, non può avere effetti collaterali. In una guardia si può verificare se un oggetto si trova in un certo stato, usando l'operatore logico `oclInState` del linguaggio OCL, ma anche questa possibilità offerta dal linguaggio va usata con parsimonia, per motivi di chiarezza e semplicità.

Macchine a stati interagenti

È possibile descrivere attività concorrenti anche senza ricorrere alla scomposizione in regioni concorrenti, quando tali attività sono eseguite da oggetti diversi, a cui sono associate macchine a stati distinte.

Le figure 4.31 e 4.32 mostrano come esempio un sistema formato da tre componenti interagenti.

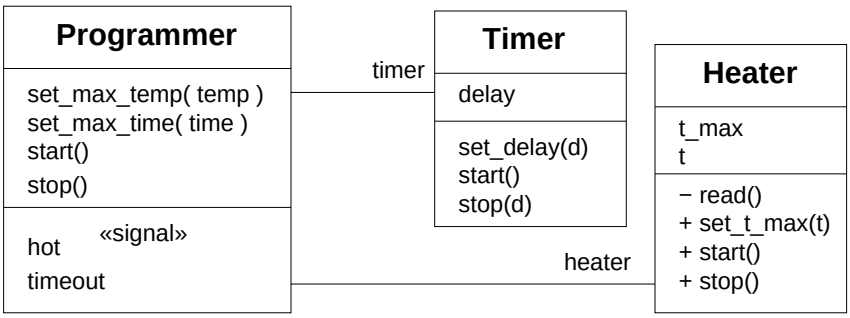


Figura 4.31: Macchine a stati interagenti (1).

Il sistema (fig. 4.31) è costituito da un timer ed una stufa controllati da un programmatore. L'utente usa il programmatore per impostare la temperatura massima e la durata del riscaldamento e per accenderlo e spengerlo. Il programmatore comanda il timer e la stufa, e riceve i segnali hot, che modella il superamento della temperatura massima, e timeout, che modella il raggiungimento della durata del riscaldamento. I due segnali accettati dal programmatore vengono dichiarati nello scompartimento marcato con lo stereotipo ≪signal≫.

La fig. 4.32 mostra le macchine a stati associate alle tre classi. Tutte e tre iniziano in uno stato di attesa, in cui rimangono quando ricevono chiamate alle operazioni di impostazione ("set_ ···"). Passano dallo stato di attesa allo stato di attività e viceversa quando ricevono le chiamate delle operazioni start e stop. Il timer manda il segnale timeout allo scadere della durata impostata, e la stufa manda il segnale hot quando la temperatura raggiunge o supera il valore massimo impostato.

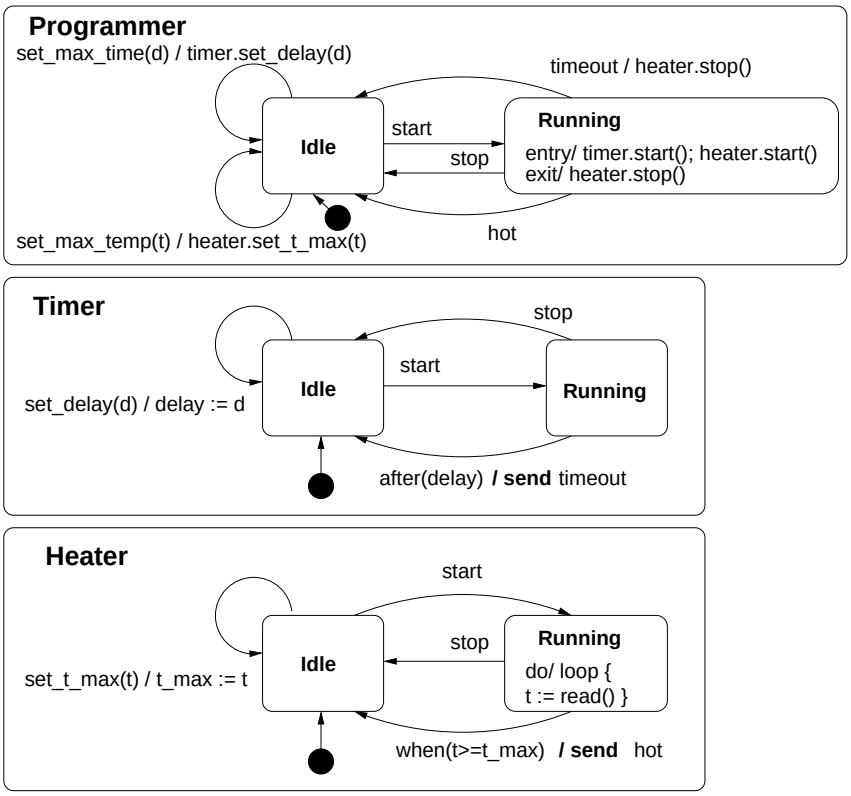


Figura 4.32: Macchine a stati interagenti (2).

4.6.8 Diagrammi di interazione

I *diagrammi di interazione* mostrano gli scambi di messaggi fra oggetti. Esistono due tipi di diagrammi di interazione: i diagrammi *di sequenza* e quelli *di comunicazione*.

Diagrammi di sequenza

Un diagramma di sequenza descrive l'interazione fra piú oggetti mettendo in evidenza il flusso di messaggi scambiati (chiamate di operazioni o segnali) e la loro successione temporale. I diagrammi di sequenza sono quindi adatti a rappresentare degli *scenari* possibili nell'evoluzione di un insieme di oggetti. È bene osservare che ciascun diagramma di sequenza rappresenta esplicitamente una o piú istanze delle possibili sequenze di messaggi, mentre un diagramma di stato definisce implicitamente tutte le possibili sequenze di messaggi ricevuti o inviati da un oggetto interagente con altri.

Un diagramma di sequenza è costituito da simboli chiamati *lifeline*, linee verticali che rappresentano l'evoluzione temporale di ciascun oggetto coinvolto nell'interazione. Ogni linea parte da un rettangolo che identifica l'oggetto. Lo scambio di un messaggio si rappresenta con una freccia dalla linea verticale dell'oggetto sorgente a quella del destinatario. L'ordine dei messaggi lungo le linee verticali ne rispecchia l'ordine temporale. La fig. 4.33

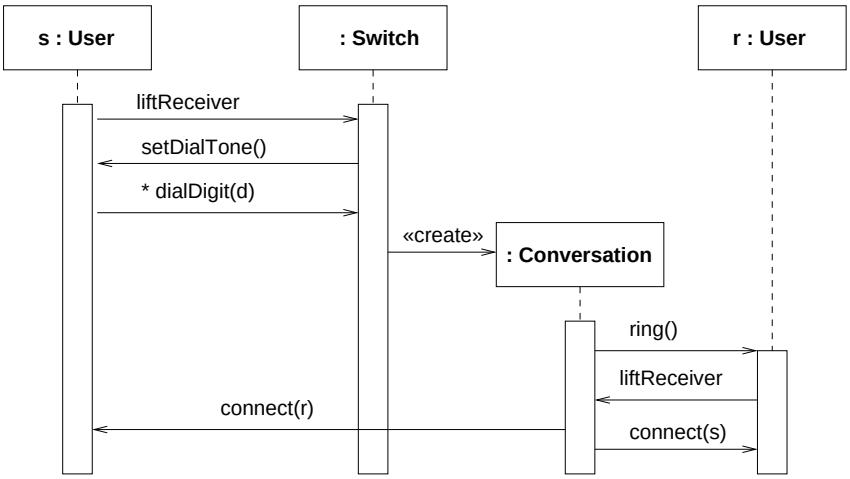


Figura 4.33: Un diagramma di sequenza.

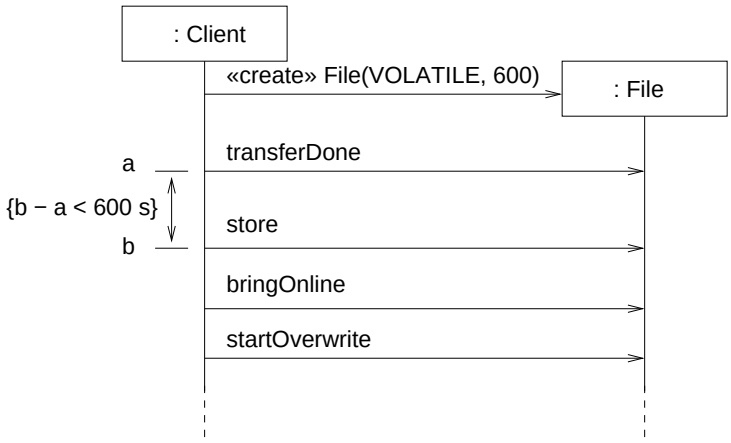


Figura 4.34: Vincoli temporali.

mostra un semplice diagramma di sequenza che descrive l'interazione di due utenti con un centralino telefonico.

Si può disegnare anche un asse dei tempi, parallelo alle lifeline, su cui evidenziare gli eventi, etichettando gli istanti corrispondenti con degli identificatori o con dei valori temporali, che possono essere usati per specificare vincoli di tempo (fig. 4.34). Inoltre, i periodi in cui un oggetto è coinvolto in un'interazione possono essere messi in evidenza sovrapponendo una striscia rettangolare alla linea verticale.

Le figure successive mostrano due possibili scenari per il sistema modellato dai diagrammi di fig. 4.31 e 4.32.

Il diagramma in fig. 4.35 modella un'evoluzione in cui la temperatura raggiunge il limite prima che avvenga un timeout o la stufa venga spenta, mentre un caso in cui avviene prima il timeout è rappresentato in fig. 4.36.

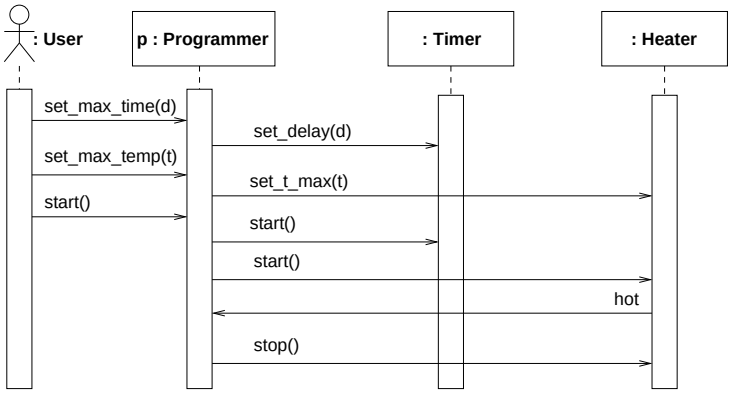


Figura 4.35: Scenario 1.

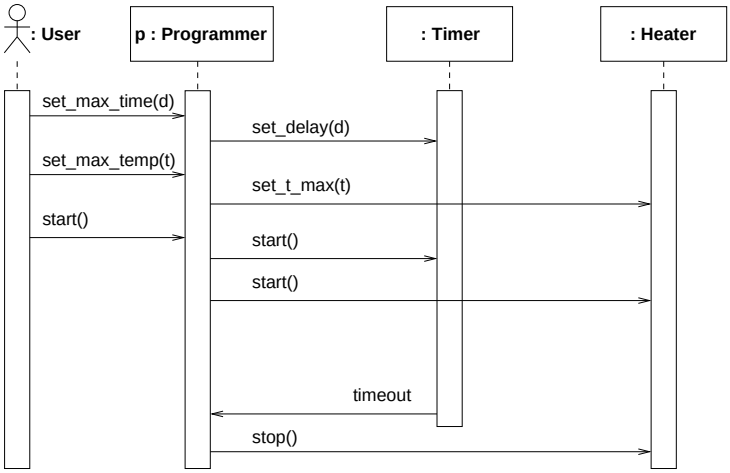


Figura 4.36: Scenario 2.

Diagrammi di comunicazione

Un diagramma di comunicazione (chiamato *diagramma di collaborazione* in UML1) mette in evidenza l'aspetto strutturale di un'interazione, mostrando esplicitamente i legami (istanze di associazioni) fra gli oggetti, e ricorrendo a un sistema di numerazione strutturato per indicare l'ordinamento temporale dei messaggi.

La fig. 4.37 mostra il diagramma di comunicazione corrispondente al diagramma di sequenza di fig. 4.33.

4.6.9 Diagrammi di attività

I *diagrammi di attività* servono a descrivere il flusso di controllo e di informazioni dei processi. In un modello di analisi si usano spesso per descrivere i processi del dominio di applicazione, come, per esempio, le procedure richieste nella gestione di un'azienda, nello sviluppo di un prodotto, o nelle transazioni economiche. In un modello di progetto

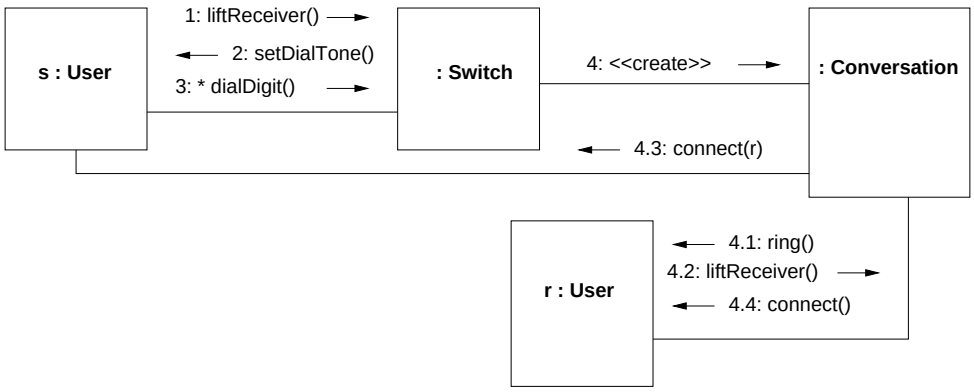


Figura 4.37: Un diagramma di comunicazione.

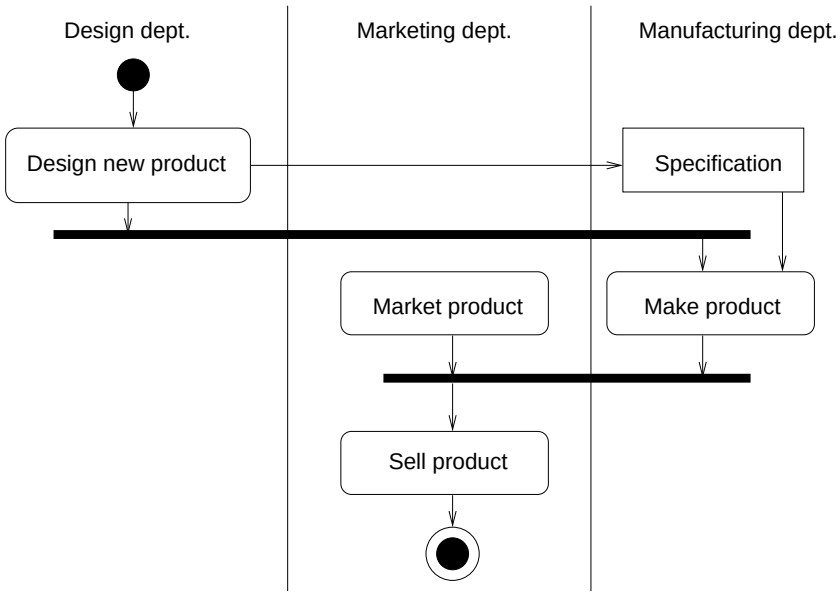


Figura 4.38: Un diagramma di attività con corsie e nodi oggetto.

possono essere usati per descrivere algoritmi o implementazioni di operazioni. Si può osservare che, nella loro forma piú semplice, i diagrammi di attività sono molto simili ai tradizionali *diagrammi di flusso* (*flowchart*).

Un diagramma di attività è formato da *nodi* e *archi*. I nodi rappresentano *attività* svolte in un processo, *punti di controllo* del flusso di esecuzione, o *oggetti* elaborati nel corso del processo. Gli archi collegano i nodi per rappresentare i flussi di controllo e di informazioni.

I diagrammi di attività possono descrivere attività svolte da entità differenti, raggruppandole graficamente. Ciascuno dei gruppi cosí ottenuti è una *partizione*, detta anche *corsia* (*swimlane*).

L'esempio in fig. 4.38 descrive (in modo molto semplificato) il processo di sviluppo di un prodotto, mostrando quali reparti di un'azienda sono responsabili per le varie attività.

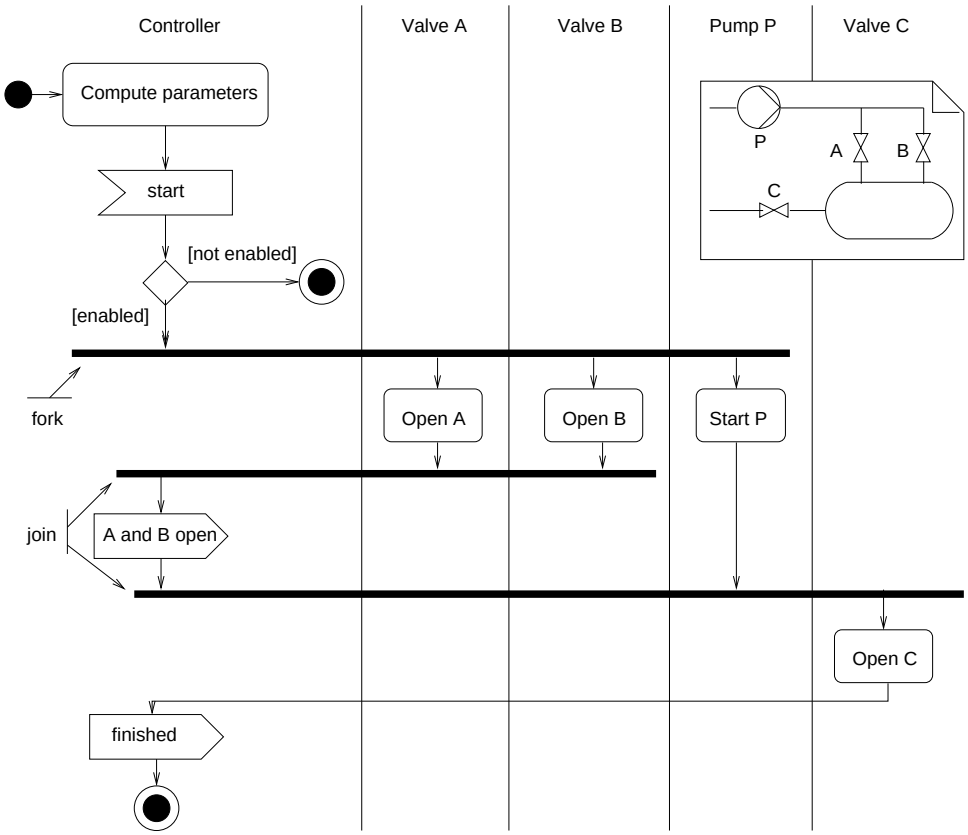


Figura 4.39: Diagramma di attività per un processo di controllo.

Il nodo Specification è un nodo oggetto, in questo caso il documento di specifica prodotto dall'attività Design.

Il diagramma di fig. 4.39 descrive un processo di controllo: alla ricezione di un segnale di *start*, se il sistema è abilitato vengono messe in funzione le valvole A e B e la pompa P. Quando tutte e due le valvole sono aperte viene emesso il segnale *A and B open*, e quando la pompa è stata avviata si apre la valvola C, e viene emesso il segnale *finished*. Osserviamo che i simboli per la ricezione e l'invio di segnali si possono usare anche nei diagrammi di stato.

Le tre linee orizzontali spesse rappresentano un nodo di controllo di tipo *fork* (diramazione del flusso di controllo in attività parallele) e due nodi di tipo *join* (ricongiungimento di attività parallele).

4.6.10 Meccanismi di estensione

In UML esistono tre meccanismi di estensione che permettono di adattare il linguaggio a esigenze specifiche: *vincoli*, *stereotipi* e *valori etichettati*.

Vincoli

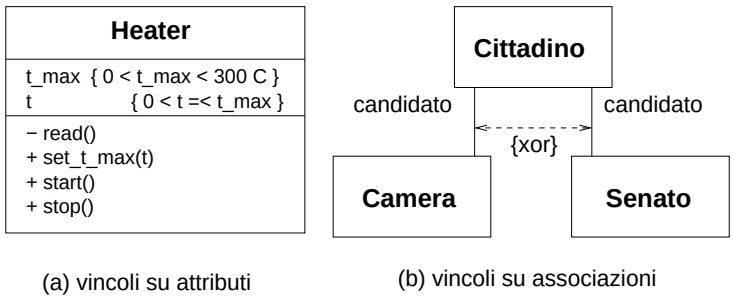


Figura 4.40: Vincoli.

I *vincoli* sono annotazioni che descrivono determinate condizioni imposte al sistema specificato, come, per esempio, l'insieme di valori ammissibili per un attributo, come in fig. 4.40(a), o il fatto che certe relazioni fra oggetti siano mutuamente esclusive, come in fig. 4.40(b), che modella l'incompatibilità fra la candidatura alla Camera e quella al Senato per uno stesso cittadino.

I vincoli possono venire espressi in linguaggio naturale, oppure in OCL, o in qualsiasi linguaggio appropriato. La sintassi UML richiede che i vincoli vengano scritti fra parentesi graffe.

Stereotipi e valori etichettati

Il *metamodello* del linguaggio UML è il suo vocabolario e la sua grammatica. Il metamodello è costituito da *metaclassi*, ognuna delle quali definisce un tipo di elementi di modello, cioè un concetto base del linguaggio. P.es., il concetto di *classe* è definito dalla metaclasse **Class**, che a sua volta dipende dalle metaclassi **Attribute**, **Operation**, eccetera.

Uno *stereotipo* è un nuovo tipo di elementi di modello, cioè una nuova metaclasse, ottenuta aggiungendo proprietà (metaattributi) e vincoli ad una metaclasse preesistente.

Applicare uno stereotipo S ad un elemento E significa che E possiede le proprietà e soddisfa i vincoli di S. L'elemento E assegna valori specifici alle proprietàdefinite in S mediante valori etichettati (*tagged values*). Si possono applicare piú stereotipi ad uno stesso elemento.

Nell'esempio di fig. 4.41, lo stereotipo **ClockType** estende **Class** con queste proprietà:

- (time) nature: discreto o denso;
- resolAttr: risoluzione temporale;
- setTime, getTime: operazioni per inizializzare e leggere la misura del tempo.

Uno sviluppatore *applica* lo stereotipo creandone un'istanza (**Chronometer**) definita dai seguenti valori etichettati:

- `{nature = discrete}`;
- `{resolAttr = resolution}`;

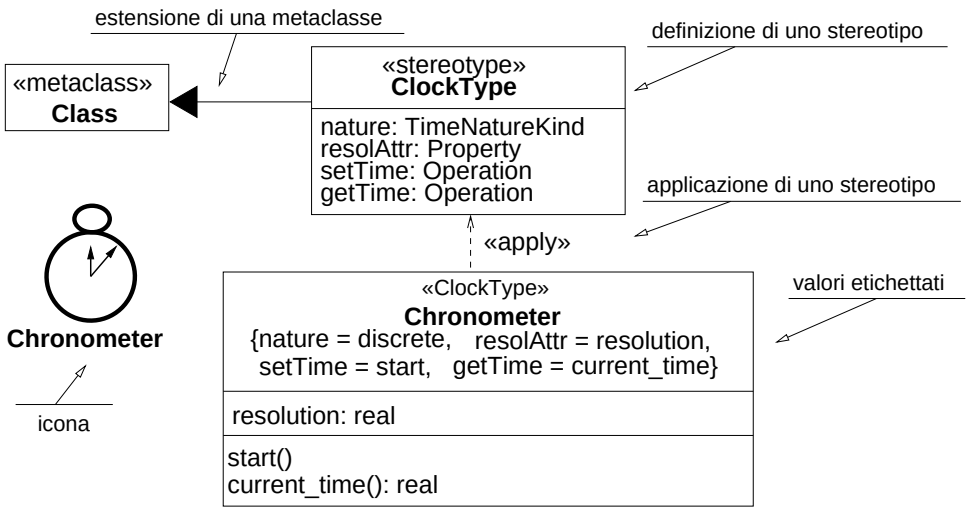


Figura 4.41: Definizione di uno stereotipo.

- `{setTime = start}`;
- `{getTime = current_time}`;

La figura mostra i simboli usati per l'estensione di metaclasse e per l'applicazione di stereotipo. Inoltre mostra la presentazione minima, usando un'icona per lo stereotipo ≪ClockType≫, della classe **Chronometer**, alternativa al rettangolo col nome dello stereotipo.

Uno stereotipo si può usare per vari scopi, fra cui:

- introdurre elementi di modello specifici di un dominio di applicazione, come nell'esempio precedente;
- mettere in evidenza il ruolo nel sistema di certe istanze di elementi di modello, come nell'esempio (a) di fig. 4.42;
- aggiungere ad istanze di elementi di modello informazioni non pertinenti al sistema modellato, ma utili per la gestione del processo di sviluppo, come nell'esempio (b);
- aggiungere informazioni che possono essere interpretate da strumenti di sviluppo, p.es. per generare codice o documentazione, come nell'esempio (c).

Profili

Un profilo è un insieme organico di stereotipi, che definisce nuovi elementi di modello utili in particolari campi di applicazione. Lo OMG cura la standardizzazione di profili proposti da consorzi di industrie ed enti di ricerca. Per esempio, il profilo UML MARTE (Modeling and Analysis of Real-time and Embedded systems[19])è stato sviluppato e standardizzato per la modellazione di sistemi embedded e in tempo reale.

La fig. 4.43 mostra schematicamente le relazioni fra un profilo, il metamodello UML ed un modello a cui si applica il profilo.

[19] `http://www.omg.org/omgmarte/`

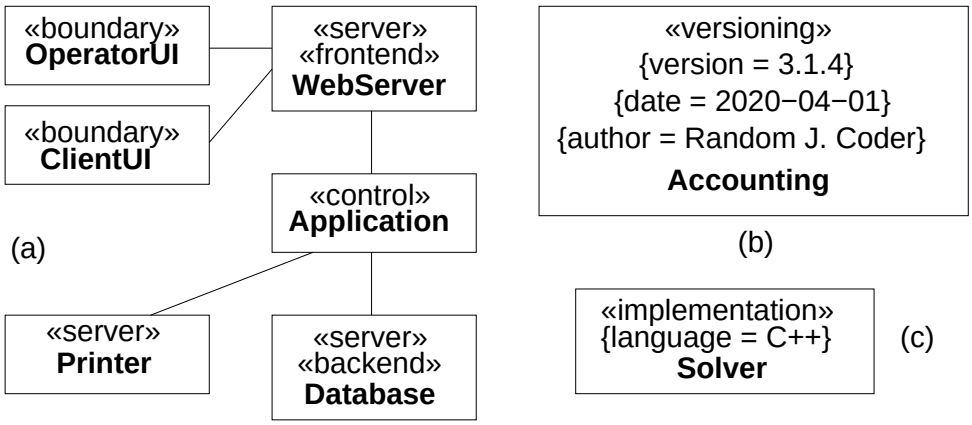


Figura 4.42: Usi degli stereotipi.

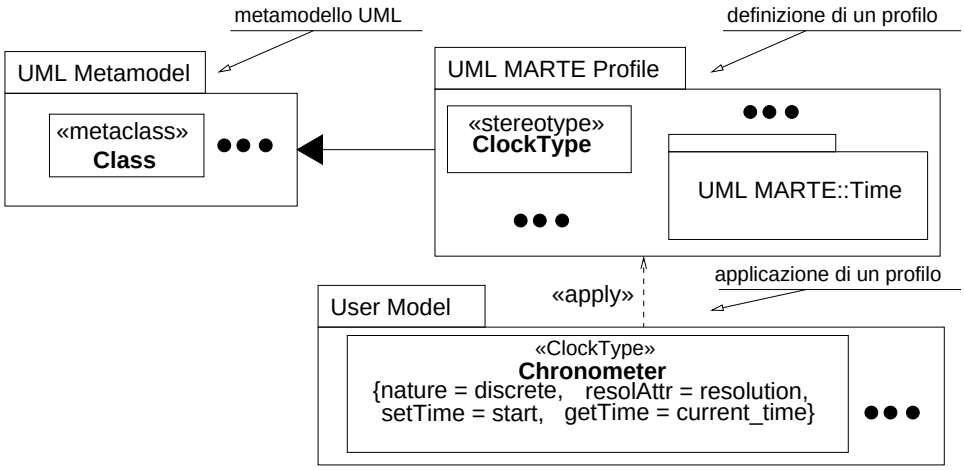


Figura 4.43: Profili.

Letture

Obbligatorie: cap. 5 e sez. 4.6 Ghezzi, Jazayeri, Mandrioli [18], oppure cap. 2 (esclusi 2.5.1, 2.5.2, pagg. 90–100 di 2.6.2, 2.6.4, 2.7.3) Ghezzi et al. [16], oppure cap. 7 (esclusi 7.4.2, 7.6.2, 7.7), sez. 8.4–8.9, sez. 23.1, sez. 24.1 Pressman [40].

Facoltative: cap. 23 Pressman [40]. Sulla logica, Cap. 1 e 2 Quine [41].

Capitolo 5

Il progetto

L'obiettivo dell'attività di progetto è produrre una *architettura software*, cioè una descrizione della struttura del sistema da realizzare, espressa in termini dei suoi *moduli*, cioè delle sue parti e delle loro relazioni reciproche. Piú precisamente, l'*architettura logica* del software è definita da:

moduli: unità funzionali, ciascuna destinata a svolgere un insieme ben delimitato di compiti offrendo dei servizi attraverso un'*interfaccia*. Dal punto di vista strutturale, i moduli si possono dividere in:

- **moduli unitari:** elementi base ("mattoni") dell'architettura, che non contengono altri moduli; nelle architetture orientate agli oggetti, le classi sono moduli unitari.
- **componenti:** formati generalmente da piú moduli unitari, sono caratterizzati da una o piú interfacce dichiarate esplicitamente per specificare i servizi offerti e richiesti e facilitare la sostituzione ed il riuso dei moduli.
- **sottosistemi:** raggruppamenti di moduli ai livelli più alti della struttura, di cui sono i blocchi principali.

relazioni: descrivono i ruoli reciproci dei moduli nell'architettura *architettura* software; i principali tipi di relazione sono:

- **associazioni:** legami strutturali fra moduli che interagiscono.
- **generalizzazione:** relazione caratteristica delle architetture orientate agli oggetti, in cui si dice che una classe A generalizza una classe B se le istanze di B godono di tutte le proprietà di A.
- **dipendenze:** un modulo A *dipende* da un modulo B se una modifica nella definizione di B richiede che venga modificato anche A; esistono molti tipi di dipendenze, fra cui la piú comune è quella di *uso*, che si ha quando un modulo richiede i servizi di un altro.

L'architettura software deve rispondere a due esigenze contrastanti: deve essere abbastanza astratta da permettere una facile comprensione del sistema (e quindi la verifica

dell'adeguatezza del progetto rispetto alle specifiche), ed abbastanza dettagliata da fornire una guida alla successiva fase di codifica.

Nel progetto dell'architettura software bisogna tener conto dell'architettura hardware, che pone dei vincoli alle scelte del progettista. Il progettista del software deve anche specificare le relazioni fra architettura software ed architettura hardware, e in particolare l'assegnamento dei vari componenti software ai componenti hardware che li devono eseguire. Si devono quindi considerare:

architettura fisica del software: l'insieme dei file richiesti per costruire il prodotto finale (attività di *build*, o *costruzione*) e per il suo funzionamento a tempo di esecuzione:

- **file sorgente:** il prodotto immediato dell'attività di codifica, costituito da file di testo scritti in un linguaggio di programmazione.
- **file oggetto e librerie:** prodotti dalla compilazione dei file sorgente.
- **file eseguibili:** prodotti dal collegamento (*linking*) dei file oggetto e delle librerie.
- **dati:** informazioni da elaborare e risultati.
- **altri:** pagine web, *script*, immagini, eccetera.

architettura dello hardware: l'insieme dei dispositivi fisici su cui viene eseguito il sistema software, con i reciproci collegamenti; i dispositivi possono essere interi calcolatori o loro parti, o periferiche di vario tipo.

L'attività di progetto è un processo iterativo attraverso una serie di approssimazioni successive a partire da un primo progetto ad alto livello, che índica una suddivisione del sistema in pochi grandi blocchi, fino ad ottenere una struttura sufficientemente dettagliata per l'implementazione. Generalmente la fase di progetto viene scomposta di due sottofasi, il *progetto architetturale*, o *di sistema*, ed il *progetto dettagliato*.

È difficile stabilire a priori a quale livello di dettaglio si debba fermare la fase di progetto. Inoltre, l'uso di metodi formali, di tecniche di prototipazione e di linguaggi ad alto livello (adatti sia al progetto che alla codifica) rende sfumata la distinzione fra progetto e codifica.

Neppure la distinzione fra la fase di analisi e specifica dei requisiti e la fase di progetto è completamente netta, poiché nella fase di progetto spesso si rilevano delle incompletezze e inconsistenze delle specifiche, che devono quindi essere riconsiderate e riformulate. Per questo sono sempre piú diffusi i processi di sviluppo che alternano ciclicamente fasi di analisi e fasi di progetto.

Nella prima parte di questo capitolo verranno trattati i concetti generali relativi all'attività di progetto, indipendentemente dal linguaggio di modellazione adottato. Questi concetti verranno ripresi nell'ultima sezione con riferimento al linguaggio UML, per modelli orientati agli oggetti.

5.1 Obiettivi della progettazione

Data una specifica, esistono molti modi per realizzarla. La scelta fra le diverse possibilità è guidata sia da vincoli di carattere economico, sia dalla necessità di conseguire un'adeguata qualità del prodotto o del progetto stesso. Per qualità del progetto si intendono sia quelle caratteristiche che, pur non essendo direttamente percepibili dall'utente, determinano la qualità del prodotto, sia quelle che offrono un vantaggio economico al produttore del software, rendendo piú efficace il processo di sviluppo.

La proprietà fondamentale del prodotto è ovviamente il rispetto dei requisiti funzionali. Alle proprietà relative alla sua qualità corrispondono i requisiti non funzionali, che in molti casi si possono trasformare in requisiti funzionali: per esempio, il requisito di usabilità si può tradurre in requisiti funzionali riguardanti l'offerta di determinate funzioni, come il back-up automatico dei dati o la possibilità di personalizzare l'interfaccia utente.

Fra le caratteristiche che determinano la qualità del prodotto o del progetto, citiamo l'*affidabilità*, la *modificabilità* (ricordiamo il principio "*design for change*"), la *comprensibilità* e la *riusabilità*. L'esperienza accumulata finora dimostra che queste proprietà dipendono fortemente da un'altra, la *modularità*. Un sistema è *modulare* se è composto da un certo numero di sottosistemi, ciascuno dei quali svolge un compito ben definito e dipende dagli altri in modo semplice. È chiaro che un sistema cosí strutturato è piú comprensibile di uno la cui struttura venga oscurata dalla mancanza di una chiara suddivisione dei compiti fra i suoi componenti, e dalla complessità delle dipendenze reciproche. La comprensibilità a sua volta, insieme alla semplicità delle interdipendenze, rende il sistema piú facile da verificare, e quindi piú affidabile. Inoltre, il fatto che ciascun sottosistema sia il piú possibile indipendente dagli altri ne rende piú facili la modifica e il riuso.

5.1.1 Strutturazione e complessità

Per spiegare in modo piú concreto il concetto di *strutturazione*, e il nesso fra struttura e complessità, consideriamo due modi diversi di organizzare un programma in C, i cui componenti potrebbero essere le funzioni che formano il programma. In un primo caso il programma (che supponiamo composto da un migliaio di istruzioni) consiste di un'unica funzione (`main()`), e nell'altro consiste di dieci o venti funzioni. Nel primo caso il programma è poco strutturato poiché tutte le istruzioni del programma stanno in un unico "calderone", ove ciascuna di esse può interagire con le altre, per esempio attraverso i valori di variabili globali; di conseguenza la complessità è alta. Nel secondo caso il programma è piú strutturato, poiché il campo di azione di ciascuna istruzione è limitato alla funzione a cui appartiene, e quindi ogni funzione "nasconde" le proprie istruzioni. Se lo scopo di ciascuna funzione è chiaro, il programma può essere descritto e compreso in termini delle relazioni fra le funzioni componenti, ed è quindi meno complesso.

Il grado di strutturazione di un sistema dipende quindi dal giusto valore di *granularità*, cioè dalle dimensioni dei componenti considerati, che non deve essere né troppo grossa né troppo fine. Dal momento che i componenti elementari del software sono le singole

istruzioni del linguaggio di programmazione usato, che hanno una granularità finissima, un software ben strutturato ha un'organizzazione gerarchica che lo suddivide in vari strati con diversi livelli di astrazione: il sistema complessivo è formato da un numero ridotto di sottosistemi, ciascuno dei quali è diviso in un certo numero di moduli, divisi a loro volta in sottomoduli, e cosí via. In questo modo, entro ciascun livello di astrazione si può esaminare una parte del sistema in termini di pochi elementi costitutivi. La fig. 5.1 cerca di visualizzare intuitivamente questi concetti; nello schema di destra nella figura, i cerchietti rappresentano le interfacce dei moduli, di cui parleremo piú oltre.

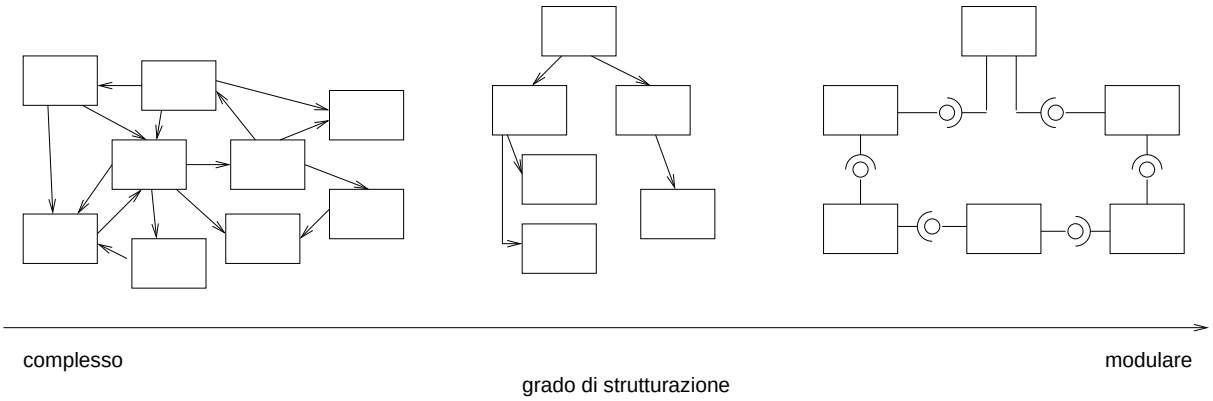


Figura 5.1: Complessità e struttura.

La modularità di un sistema influenza anche la pianificazione e la gestione dell'attività di progetto e di codifica, poiché rende possibile la ripartizione del lavoro fra diversi gruppi di sviluppatori, i quali possono lavorare in modo tanto piú indipendente (anche dal punto di vista organizzativo: si pensi a progetti sviluppati da consorzi di aziende diverse, magari situate in paesi diversi) quanto piú i sottosistemi sono logicamente indipendenti.

È quindi fondamentale, nel progettare un sistema, conoscere e comprendere i princípi ed i metodi che permettono di realizzare architetture modulari.

5.2 Moduli

Il termine *modulo* può assumere significati diversi in contesti diversi, e in questo corso adotteremo il significato piú generale: una porzione di software che contiene e fornisce risorse o servizi[1], e a sua volta può usare risorse o servizi offerti da altri moduli. Un modulo viene quindi caratterizzato dalla sua *interfaccia*, cioè dall'elenco dei servizi offerti (o *esportati*) e richiesti (o *importati*). Queste due parti dell'interfaccia si chiamano *interfaccia offerta* e *interfaccia richiesta*, ma spesso la parola "interfaccia" viene usata in riferimento alla sola parte offerta.

Le risorse offerte da un modulo possono essere strutture dati, operazioni, e definizioni di tipi. L'interfaccia di un modulo può specificare un *protocollo*, cioè un insieme di vincoli

[1] Useremo questi due termini come sinonimi.

sulle possibili sequenze di scambi di messaggi (o chiamate di operazioni) fra il modulo e i suoi clienti. Alle operazioni si possono associare *precondizioni* e *postcondizioni* (v. oltre). Infine, l'interfaccia può specificare le *eccezioni*, cioè condizioni anomale che si possono verificare nell'uso del modulo, e le azioni richieste per gestire tali condizioni, implementate da *gestori* (*handler*). In generale, un modulo può anche offrire l'accesso diretto (cioè senza la mediazione di operazioni apposite) alle proprie strutture dati, ma questa pratica è sconsigliata perché, come verrà illustrato, va a detrimento della modularità.

L'interfaccia di un modulo è una specifica, che viene realizzata dall'*implementazione* del modulo. Possiamo distinguere fra implementazioni *composte*, in cui l'interfaccia del modulo viene implementata per mezzo di piú sottomoduli, e implementazioni *semplici*, in cui l'implementazione (costituita da definizioni di dati e operazioni in un qualche linguaggio di programmazione) appartiene al modulo stesso. La fig. 5.2 riassume schematicamente la definizione di modulo qui esposta.

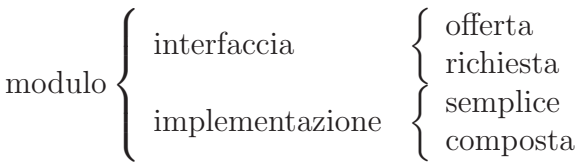


Figura 5.2: Modulo logico.

L'interfaccia e l'implementazione definiscono un *modulo logico*, cioè un'entità astratta capace di interagire con altre entità. Nella fase di codifica vengono prodotti dei file contenenti codice sorgente nel linguaggio di programmazione adottato. Chiameremo *moduli fisici* o *artefatti* sia i file sorgente, sia i file collegabili ed eseguibili ottenuti per compilazione e collegamento. I moduli fisici contengono (in linguaggio sorgente o in linguaggio macchina) le definizioni delle entità che realizzano i moduli logici, cioè strutture dati e sottoprogrammi.

La corrispondenza fra moduli logici e fisici dipende sia dal linguaggio e dall'ambiente di programmazione che da scelte fatte dagli sviluppatori. Questa corrispondenza non è necessariamente biunivoca, in quanto un modulo fisico può contenere le definizioni relative a piú moduli logici (per esempio, un file di libreria è un modulo fisico contenente le implementazioni di piú moduli logici), oppure le definizioni appartenenti a un modulo logico possono essere suddivise fra piú moduli fisici (per esempio, un sottosistema è un modulo logico la cui implementazione può essere divisa fra piú file sorgente). Ai fini della mantenibilità e della modificabilità del sistema, è importante che la corrispondenza fra struttura logica e struttura fisica sia chiara e razionale. Una buona organizzazione dei moduli fisici permette un uso piú efficace di strumenti automatici di programmazione (come, per esempio il programma `make`) e di gestione delle configurazioni (per esempio, i sistemi *git* ed *SVN*).

La maggior parte dell'attività di progetto è rivolta alla definizione della struttura logica del sistema, ma nelle fasi piú vicine all'implementazione si può definire, almeno a grandi linee, anche la struttura fisica. Nel séguito ci riferiremo quasi esclusivamente alla struttura logica.

5.2.1 Interfaccia e implementazione

Per progettare l'interfaccia di un modulo bisogna compiere un insieme di scelte guidate dai princípi della *divisione delle responsabilità* e dell'*information hiding* (occultamento dell'informazione) (Parnas [36][2]).

La divisione delle responsabilità

Il principio della divisione delle responsabilità dice che bisogna cercare di suddividere il lavoro svolto dal sistema (e ricorsivamente da ciascun sottosistema) fra i vari moduli, in modo che a ciascuno di essi venga affidato un compito ben definito e limitato (ricordiamo il vecchio motto del sistema operativo Unix: "*do one thing well*"). I moduli progettati secondo questo criterio hanno la proprietà della *coesione*, cioè di offrire un insieme omogeneo di servizi. Per esempio, un modulo che calcola dei valori e provvede anche a scriverli sull'output è poco coeso, poiché la rappresentazione dei risultati è indipendente dal procedimento di calcolo: per esempio, un insieme di risultati potrebbe essere visualizzato con una tabella oppure un grafico, o non essere visualizzato affatto, essendo destinato ad ulteriori elaborazioni, È meglio quindi che la funzione di produrre un output venga affidata a un modulo specializzato, in modo da avere un sistema piú strutturato e flessibile.

Precondizioni e postcondizioni La divisione di responsabilità comporta anche la necessità di specificare gli obblighi reciproci dei moduli. Un modulo fornitore di un servizio garantisce ad ogni modulo cliente che, in conseguenza dell'espletamento del servizio (per esempio, l'esecuzione di una procedura), saranno vere certe relazioni logiche (*postcondizioni*) sullo stato del sistema. Il modulo fornitore, però, si aspetta che all'atto dell'invocazione del servizio siano vere altre relazioni (*precondizioni*). Per esempio, un modulo che deve calcolare la radice quadrata y di un numero x ha come precondizione un vincolo sul segno del valore di x ($x \geq 0$), e come postcondizione una relazione fra x e il risultato ($x = y^2$). Le post-condizioni e le pre-condizioni quindi rappresentano, rispettivamente, le responsabilità del modulo fornitore e quelle del modulo cliente riguardo a un dato servizio. È buona prassi scrivere le precondizioni e postcondizioni di ciascuna operazione nella documentazione di progetto e nel codice sorgente.

Gestione degli errori Un altro aspetto importante della divisione delle responsabilità è la gestione degli errori e delle situazioni anomale. Se si prevede che nello svolgimento di un servizio si possano verificare delle situazioni anomale, bisogna decidere se tali situazioni possono essere gestite nel modulo fornitore, nascondendone gli effetti ai moduli clienti, oppure se il modulo fornitore deve limitarsi a segnalare il problema, delegandone la gestione ai moduli clienti.

[2] `http://www.ing.unipi.it/~a009435/issw/extra/criteria.pdf`

Information hiding

Il principio dell'*information hiding* afferma che bisogna rendere inaccessibile dall'esterno tutto ciò che non è strettamente necessario all'interazione con gli altri moduli, in modo che vengano ridotte al minimo le dipendenze e quindi sia possibile progettare, implementare e collaudare ciascun modulo indipendentemente dagli altri.

Dopo che è stato individuato il compito di un modulo, si vede che tale compito, per essere svolto, ha bisogno di varie strutture dati e di operazioni (o di altre risorse, magari piú astratte, come tipi di dati o politiche di gestione di certe risorse). Il progettista deve decidere quali di queste entità devono far parte dell'interfaccia, ed essere cioè accessibili dall'esterno. Le entità che non sono parte dell'interfaccia servono unicamente a implementare le altre, e non devono essere accessibili. Questa scelta non è sempre facile e immediata, poiché spesso può sembrare che certe parti dell'implementazione (per esempio, una procedura) possano essere utili ad altri moduli. Comunque, quando si è deciso che una certa entità fa parte dell'implementazione bisogna che questa sia effettivamente nascosta, poiché la dipendenza di un modulo cliente dall'implementazione di un modulo fornitore fa sí che quest'ultimo non possa venire modificato senza modificare il cliente. In un sistema di media complessità il costo di qualsiasi cambiamento può diventare proibitivo, se le dipendenze reciproche costringono gli sviluppatori a propagare i cambiamenti da un modulo all'altro.

In particolare, conviene nascondere nell'implementazione le strutture dati, a cui si dovrebbe accedere soltanto per mezzo di sottoprogrammi. Anche le politiche di accesso a una risorsa gestita da un modulo (per esempio, ordinamento FIFO oppure LIFO delle richieste di accesso), di norma devono essere nascoste ai moduli clienti, in modo che questi non dipendano da tali politiche.

In conclusione, nello specificare un'architettura software si cerca di ottenere il massimo *disaccoppiamento* fra i moduli.

5.2.2 Relazioni fra moduli

Come già detto, un'architettura software è la specifica di un insieme di moduli e delle loro relazioni. Fra queste, hanno un'importanza particolare le relazioni di *composizione* e di *uso*.

La relazione di composizione sussiste fra due moduli quando uno è parte dell'altro ed è necessariamente gerarchica, cioè viene descritta da un grafo orientato aciclico.

La relazione di uso sussiste quando il corretto funzionamento di un modulo richiede la presenza e generalmente[3] il corretto funzionamento di un altro modulo. Questa relazione

[3] Il corretto funzionamento del modulo usato non è sempre richiesto, poiché un modulo in certi casi contiene solo risorse "statiche", come strutture dati o definizioni di tipi, per le quali non si può parlare di "funzionamento".

permette l'esistenza di cicli nel grafo ad essa associato, però si cerca di evitare i cicli poiché i moduli che si trovano su un ciclo non possono venire esaminati e verificati isolatamente.

Le relazioni di composizione e uso sono il minimo indispensabile per definire un'architettura, ma ne esistono e se ne possono concepire molte, che possono essere piú o meno utili a seconda del tipo di sistema progettato, della metodologia di progetto, e del livello di dettaglio richiesto in ciascuna fase della progettazione. Per esempio, nei metodi orientati agli oggetti è particolarmente importante la relazione di generalizzazione o eredità. Un'altra relazione importante è la *comunicazione* fra moduli. In generale, si mettono in evidenza vari tipi di *dipendenza*, di cui l'uso è un caso particolare.

5.2.3 Tipi di moduli

Il concetto di modulo visto finora è molto generale. In questa sezione esamineremo alcuni concetti relativi a diversi tipi di componenti modulari del software.

Le *astrazioni procedurali*, il tipo di moduli piú semplice, sono moduli che non gestiscono strutture dati, ma forniscono soltanto delle procedure per calcolare valori, non necessariamente numerici. Un esempio tipico di astrazioni procedurali sono le tradizionali librerie matematiche, o quelle per elaborare stringhe di caratteri.

Un modulo che gestisce una struttura dati è chiamato *oggetto astratto*: la struttura dati è astratta in quanto vi si può accedere soltanto attraverso le operazioni definite dall'interfaccia, e la sua realizzazione concreta è nascosta. Gli oggetti astratti permettono di controllare gli accessi alle informazioni contenute in una struttura dati che devono essere condivise fra piú moduli.

Un'altra categoria di moduli è quella dei *tipi di dati astratti* (TDA, ADT), che definiscono insiemi di oggetti astratti che hanno la stessa interfaccia e la stessa implementazione. Per esempio, i numeri complessi si possono rappresentare come un TDA definito dalle operazioni dell'aritmetica complessa. L'implementazione di un numero complesso sarà una coppia di numeri che rappresentano la parte reale e la parte immaginaria oppure il modulo e l'argomento. Questa coppia può essere rappresentata da due membri di una struttura o di una classe, oppure da due componenti di un array. I due numeri, a loro volta, possono essere implementati da numeri in virgola mobile di varia precisione, da coppie di interi (*mantissa*, *esponente*), da stringhe di byte, e in altri modi ancora.

Se la struttura o l'algoritmo di un oggetto astratto (o tutti e due) possono essere resi parametrici rispetto a qualche loro caratteristica, si ottiene un *oggetto generico*. Generalmente la caratteristica che si rende generica è il tipo di qualche componente dell'oggetto. Un oggetto generico è la definizione di un insieme di oggetti che hanno strutture ed operazioni simili, ma specializzate secondo alcuni parametri. Per esempio, una tabella generica può rappresentare una famiglia di tabelle che differiscono per il tipo degli elementi o per il loro numero, o per tutte e due le caratteristiche.

Anche un tipo di dati astratto può essere reso generico, ottenendo un *tipo di dati astratto generico*.

Gli *oggetti* e le *classi* sono concetti che abbiamo già incontrato nell'àmbito delle metodologie di specifica orientate agli oggetti (sez. 4.6), dove vengono usati per rappresentare entità del dominio di applicazione. Nel progetto del software, gli oggetti sono gli elementi software che implementano l'applicazione. Alcuni di questi oggetti simulano le entità del dominio implementandone direttamente attributi e operazioni, altri partecipano al funzionamento dell'applicazione fornendo i meccanismi necessari.

Le classi dei linguaggi orientati agli oggetti permettono anche di implementare i tipi di dati astratti: per esempio, i numeri complessi sono un tipo di dati astratto definito dalle sue proprietà matematiche, che può essere implementato da una classe avente come attributi la parte reale e la parte immaginaria, oppure modulo e argomento, e come operazioni le varie operazioni dell'aritmetica complessa.

Esistono dei moduli il cui scopo è di raggruppare altri elementi (che possono essere moduli a loro volta) e di presentarli come un'unità logica. Generalmente si usa il termine *package* per riferirsi a un insieme di entità logicamente correlate (per esempio un gruppo di classi o di sottoprogrammi). Il concetto di "package" di solito implica quello di *spazio di nomi* (*namespace*): uno spazio di nomi serve a identificare un insieme di entità e ad evitare conflitti di nomenclatura fra tali entità e quelle appartenenti ad un altro spazio di nomi: due entità aventi lo stesso nome ma appartenenti a spazi di nomi diversi sono entità distinte.

Un *sottosistema* è una parte di un sistema piú grande, relativamente autonoma e caratterizzata da una propria funzione, e quindi da un'interfaccia.

Infine, accenniamo al concetto di *componente*. Questo viene generalmente inteso come un elemento software interamente definito dalle interfacce offerte e richieste, che possa essere sostituito da un'altro componente equivalente anche se diversamente implementato, e che possa essere riusato in contesti diversi, analogamente ai componenti elettronici. Secondo questa definizione, qualsiasi modulo realizzato rispettando rigorosamente il principio dell'information hiding e capace di offrire servizi di utilità abbastanza generale si potrebbe considerare un componente, però spesso si usa questo termine in modo piú restrittivo, richiedendo che i componenti si possano sostituire a tempo di esecuzione: questo impone dei particolari requisiti sugli ambienti di programmazione e di esecuzione, e richiede che i componenti vengano realizzati rispettando determinate convenzioni. Una terza definizione, intermedia fra le precedenti, richiede che i componenti seguano certe convenzioni, ma non che siano necessariamente istanziabili a tempo di esecuzione.

5.3 Linguaggi di progetto

La fase di progetto produce alcuni documenti che descrivono l'architettura del sistema. Tali documenti sono scritti in parte in linguaggio naturale (possibilmente secondo qualche standard di documentazione che stabilisca la struttura dei documenti) e in parte mediante notazioni di progetto testuali e grafiche.

Le notazioni testuali descrivono ciascun modulo usando un linguaggio simile ai linguaggi di programmazione, arricchito da costrutti che specificano le relazioni fra i moduli e descrivono precisamente le interfacce, privo di istruzioni eseguibili. Le notazioni grafiche rappresentano in forma di diagrammi i grafi definiti dalle relazioni.

Ogni linguaggio di progetto ha un suo vocabolario e le sue convenzioni per esprimere i concetti usati nello sviluppo del software. I linguaggi di programmazione, a loro volta, hanno dei costrutti linguistici che si prestano piú o meno bene a rappresentare tali concetti. Per esempio, il C non ha dei costrutti che definiscano esplicitamente un modulo, però permette di definire degli oggetti astratti sfruttando le regole di visibilità e di collegamento. In C, un modulo può essere implementato da una unità di compilazione, l'interfaccia di un modulo può essere definita da un file *header*, la relazione di uso può essere rappresentata dalle direttive `#include`. Non si possono definire dei veri e propri dati astratti, ma i programmatori possono attenersi ad una "*disciplina*" nell'uso di tipi derivati, cioè accedere alle strutture dati che implementano un tipo solo attraverso le operazioni previste per quel tipo, rinunciando a sfruttare la possibilità, concessa dal linguaggio, di accedere ai dati direttamente.

I linguaggi di programmazione piú evoluti permettono una rappresentazione piú diretta dei concetti relativi alla programmazione modulare. In ogni caso, la conoscenza dei costrutti linguistici con cui si rappresentano i vari concetti è necessaria per verificare la correttezza dell'implementazione rispetto al progetto e per comprendere la corrispondenza fra architettura logica ed architettura fisica.

Fra i linguaggi di progetto testuali possiamo citare il *CORBA Interface Definition Language* [21, 39] (IDL), usato per definire le interfacce di moduli per sistemi distribuiti[4].

Il linguaggio UML, già visto nel capitolo relativo ai linguaggi di specifica orientati agli oggetti, è anche un linguaggio grafico di progetto. Nella descrizione di un'architettura si possono usare gli stessi concetti (e le relative notazioni) usate in fase di specifica dei requisiti, ma con un piú fine livello di dettaglio: per esempio, in fase di progetto si devono specificare precisamente il numero e i tipi degli argomenti di operazioni di cui, in fase di specifica, si era dato solo il nome. Inoltre l'UML fornisce delle notazioni adatte a esprimere dei concetti propri della descrizione architetturale, come la struttura fisica dello hardware e del software. L'uso dell'UML nell'attività di progetto verrà trattato nella sez. 5.4.

5.4 Moduli in UML

In questa sezione vengono introdotte le notazioni UML per i concetti fondamentali del progetto orientato agli oggetti.

[4] `http://www.ing.unipi.it/~a009435/issw/extra/corba0708.pdf`

5.4.1 Incapsulamento e raggruppamento

Come già osservato, un programma consiste in un insieme di istruzioni che, in un certo linguaggio, definiscono numerose entità elementari, come variabili, funzioni, e tipi. Ovviamente il programma non deve essere una massa amorfa e indifferenziata di definizioni: queste definizioni devono essere raggruppate in modo da corrispondere ai moduli previsti dall'architettura, quindi un linguaggio di progetto deve fornire, prima di tutto, dei mezzi per separare e raggruppare le entità appartenenti ai moduli. Inoltre in ciascun modulo bisogna separare l'interfaccia dall'implementazione, cioè la parte visibile da quella invisibile, quindi i linguaggi devono fornire anche i mezzi per specificare quali dichiarazioni sono visibili all'esterno e quali no. Il termine *incapsulamento* si usa spesso per riferirsi alla separazione, al raggruppamento e all'occultamento selettivo delle entità che formano un modulo: definire un modulo è come chiudere i suoi componenti in una capsula (o un pacchetto) che li protegge da un uso scorretto.

Classi

Come già accennato, le classi in UML si possono usare per definire dei moduli, che possono facilmente essere implementati con i corrispondenti costrutti dei linguaggi di programmazione orientati agli oggetti. In fase di progetto si specificano completamente gli attributi e le operazioni (se non si è già fatto nella fase di analisi), indicando tipi, argomenti, valori restituiti e visibilità. Le operazioni da specificare sono quelle che definiscono l'interfaccia, e quindi hanno visibilità *pubblica* (interfaccia verso ogni altra classe) o *protetta* (interfaccia verso le classi derivate). Le operazioni con visibilità *privata* generalmente vengono aggiunte nella fase di codifica, occasionalmente nella fase di progetto dettagliato.

In fase di progetto non si definisce l'implementazione delle classi, ma in generale si sottintende che ogni operazione debba essere implementata da un metodo, sia cioè *concreta*. Spesso però è utile dichiarare in una classe delle operazioni che non devono essere implementate nella classe stessa, ma in classi derivate; queste operazioni sono dette *operazioni astratte*. Una classe che contiene almeno una operazione astratta non può, ovviamente, essere istanziata direttamente: una tale classe si dice *classe astratta* e può solo servire da base di altre classi. Le classi derivate (direttamente o indirettamente) da una classe astratta forniscono le implementazioni (*metodi*) delle operazioni astratte, fino ad ottenere delle classi *concrete*, cioè fornite di un'implementazione completa e quindi istanziabili. In UML le operazioni e le classi astratte sono caratterizzate dalla proprietà abstract. Graficamente, questa proprietà può essere evidenziata scrivendo in caratteri obliqui il nome dell'elemento interessato.

Interfacce, dipendenza e realizzazione

In UML, un'interfaccia si può rappresentare separatamente dall'entità che la implementa (o *dalle* entità, poiché un'interfaccia può essere implementata da moduli diversi). Un'interfaccia UML è un elemento di modello costituito da un elenco di operazioni, ovviamente

astratte e con visibilità pubblica, a cui si possono aggiungere dei vincoli e un protocollo. Un'interfaccia può anche contenere delle dichiarazioni di attributi: questo significa che le classi che realizzano l'interfaccia devono rendere disponibili i valori di tali attributi, ma non necessariamente implementarli direttamente sotto forma di membri dato (i loro valori, per esempio, potrebbero essere calcolati di volta in volta).

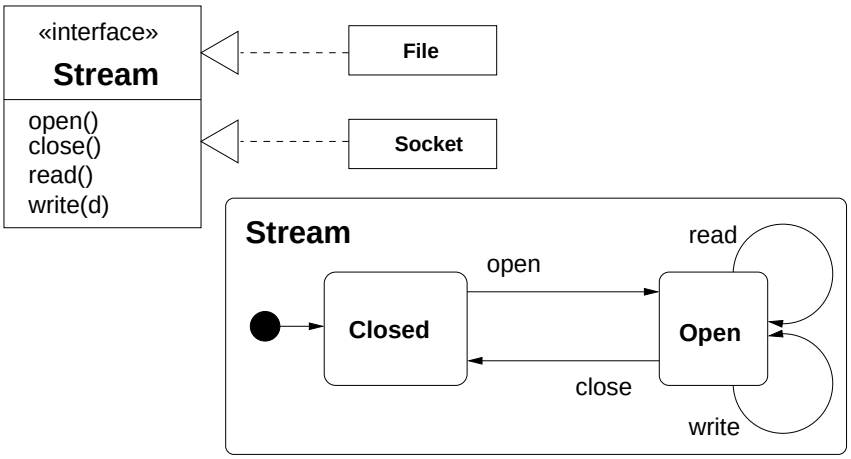


Figura 5.3: Interfacce in UML (1).

Con la notazione completa, un'interfaccia si rappresenta con un simbolo simile a quello delle classi, etichettato con parola chiave[5] ≪interface≫, come nell'esempio illustrato in fig. 5.3, in cui l'interfaccia **Stream** definisce operazioni implementate dalle classi **File** e **Socket**. La figura mostra anche una macchina a stati che specifica il protocollo che deve essere osservato dalle implementazioni di **Stream**.

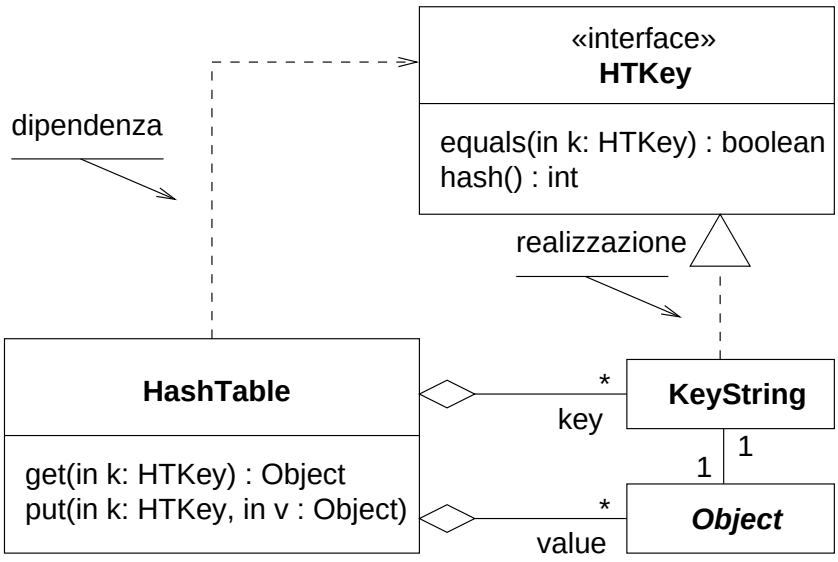


Figura 5.4: Interfacce in UML (2).

[5]Le parole chiave servono a distinguere diversi tipi di elementi di modello, mentre gli stereotipi sono specializzazioni di altri elementi di modello. Hanno la stessa sintassi.

Un altro esempio è mostrato in fig. 5.4. In questo esempio la classe **HashTable** rappresenta una tabella hash. Ricordiamo che questo tipo di tabella memorizza gli elementi (coppie $\langle chiave, valore \rangle$) in un vettore, e l'índice associato alla chiave di ciascun elemento viene calcolato in funzione del valore della chiave stessa per mezzo di una funzione detta di *hash*. Poiché la funzione di hash può associare lo stesso índice a elementi distinti, a ciascun índice corrisponde in generale non un solo elemento, ma una lista di elementi, o *bucket*. L'inserimento o la ricerca di un elemento richiedono quindi una scansione della lista individuata dall'índice restituito dalla funzione di hash. Questa ricerca a sua volta richiede un'operazione di confronto.

La funzione di hash e l'operazione di confronto dipendono dalla struttura degli elementi da memorizzare: si hanno diverse implementazioni a seconda che gli elementi siano, per esempio, stringhe di caratteri o immagini digitali. Se vogliamo far sí che la classe **HashTable** sia indipendente dal tipo di elementi che deve contenere, si ricorre al principio di divisione delle responsabilità: alla tabella di hash spetta il compito di ordinare e ricercare gli elementi in base ai risultati della funzione di hash e dell'operazione di confronto, mentre agli elementi spetta il compito di fornire l'implementazione di queste due operazioni.

L'insieme delle operazioni richieste da **HashTable** per il suo funzionamento (interfaccia richiesta) è definito dall'interfaccia **HTKey**, contenente le operazioni equals (per confrontare due elementi) e hash (per calcolare la funzione di hash). Questa relazione fra **HashTable** e **HTKey** è una *dipendenza*, rappresentata dalla freccia tratteggiata. Gli elementi che vogliamo memorizzare possono appartenere a qualsiasi classe che *realizzi* tali operazioni, cioè che comprenda le operazioni di **HTKey** nella propria interfaccia offerta, insieme ad eventuali altre operazioni. Nell'esempio, la classe **KeyString** implementa queste due operazioni, e la relazione fra **KeyString** e **HTKey** è una *realizzazione*, rappresentata da una linea tratteggiata terminante con una punta a triangolo equilatero.

Le interfacce si rappresentano in forma ridotta per mezzo di un cerchio, come mostrato in fig. 5.5. In questo caso la classe (o altro elemento di modello) che realizza l'interfaccia viene collegata a quest'ultima per mezzo di una linea continua, mentre una classe che richiede l'interfaccia viene collegata all'interfaccia con una dipendenza (freccia tratteggiata), o col simbolo di connettore *assembly* introdotto nell'UML2.

Package

L'UML dispone di un elemento di modello, chiamato *package*, per esprimere il raggruppamento di altri elementi di modello. Questi elementi possono essere di qualsiasi genere, anche interi diagrammi, e l'organizzazione in package di un modello UML può non avere una corrispondenza diretta con la struttura dell'architettura software (per esempio, il progettista potrebbe usare un package per raccogliere tutti i diagrammi di classi, un altro per i diagrammi di stato, e cosí via), anche se molto spesso i package vengono usati in modo da rispecchiare tale struttura.

L'interfaccia di un package consiste nell'insieme dei suoi elementi *esportati*, cioè resi visibili. La fig. 5.6 mostra il package **Simulator**, che contiene gli elementi pubblici (cioè

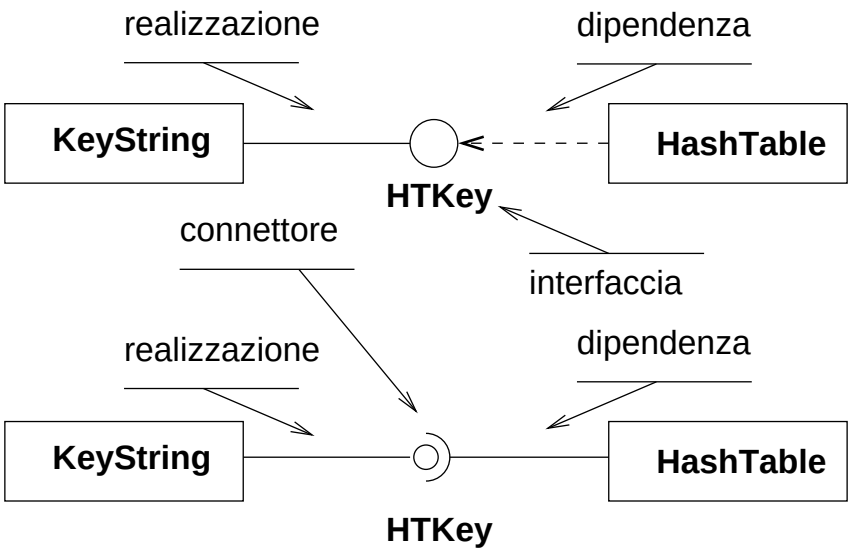


Figura 5.5: Interfacce in UML (3).

esportati) **SignalGenerators** e **Filters** (che in questo caso supponiamo essere dei package a loro volta, ma potrebbero essere classi o altri elementi di modello), e l'elemento privato **Internals**. I caratteri '+' e '-' denotano, rispettivamente, gli elementi pubblici e quelli privati.

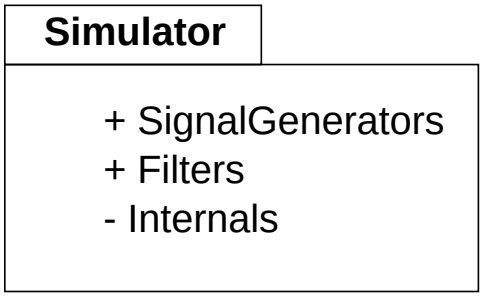


Figura 5.6: Package in UML (1)

Un package costituisce lo spazio dei nomi degli elementi contenuti. Ogni elemento appartenente ad un package ha un *nome qualificato*, formato dal nome del package e dal nome semplice dell'elemento, separati dai caratteri '::'. Se il package è contenuto in un altro package, il nome del package esterno precede quello del package interno. Se, nell'esempio di fig. 5.7, supponiamo che il package **Simulator** contenga una classe **Solver**, il nome qualificato di quest'ultima sarebbe **Simulator::Solver**. Questo nome qualificato può essere usato nelle classi appartenenti ad altri package, per esempio nella dichiarazione di parametri. Per usare soltanto il nome semplice, un altro package (nell'esempio, **UserInterface** deve *importare* il package **Simulator**, cioè inserire lo spazio di nomi di **Simulator** nel proprio). La figura mostra i tre package con le rispettive dipendenze di ≪import≫. Un'altra dipendenza mostrata comunemente è quella di uso, che si ha quando qualche elemento di un package usa elementi dell'altro.

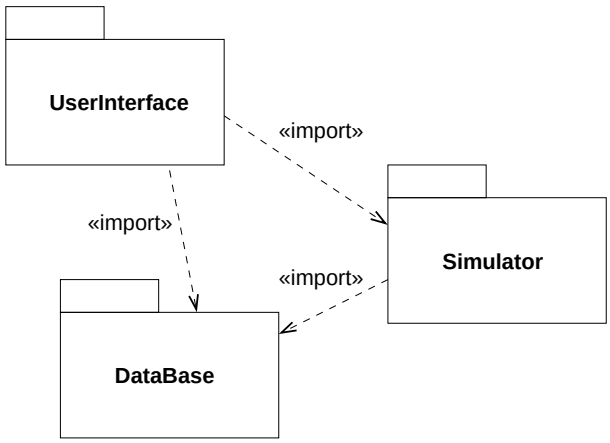


Figura 5.7: Package in UML (2)

Componenti

Un *componente* è un modulo logico definito da una o piú interfacce offerte e da una o piú interfacce richieste, sostituibile e riusabile.

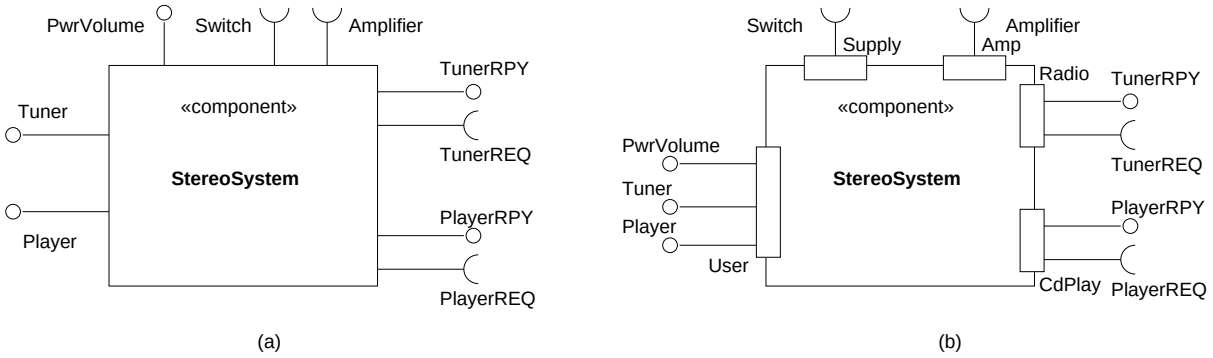


Figura 5.8: Componenti in UML.

Un componente può realizzare piú di una interfaccia: per esempio, un componente destinato a controllare un impianto stereo potrebbe avere un'interfaccia per il controllo della radio (con le operazioni scegli_banda(), scegli_canale()...), una per il lettore CD (con scegli_traccia(), successivo()...) e cosí via (fig. 5.8 (a)). Altrettanto vale per le interfacce richieste. La possibilità di suddividere la specifica di un componente in diverse interfacce permette di esprimere piú chiaramente le dipendenze fra componenti.

Generalmente le interfacce vengono associate a dei *port*, definiti come punti di interazione del componente con l'ambiente esterno. Un port è quindi caratterizzato da una o piú interfacce coinvolte in una determinata interazione. Nella fig. 5.8 (b), le interfacce PwrVolume, Tuner e Player, destinate all'interazione con l'utente, sono raggruppate nel port User, mentre le coppie di interfacce TunerREQ e TunerRPY, e PlayerREQ e PlayerRPY, usate rispettivamente per comunicare con la radio e col lettore di CD, sono assegnate

ai port Radio e CdPlay. Gli altri due port riguardano l'interazione con l'alimentazione e l'amplificatore.

Un componente può essere realizzato *direttamente* da un'istanza di una classe, o *indirettamente* da istanze di piú classi o componenti cooperanti; nel secondo caso è possibile mostrare la struttura interna del componente, come mostra la fig. 5.9. In questo esempio supponiamo che il componente venga implementato dalle istanze di alcune classi. Le frecce fra i port e le interfacce delle classi rappresentano la relazione di *delega*, ed è possibile etichettarle esplicitamente con la parola chiave ≪delegate≫. Questa relazione mostra la provenienza o la destinazione delle comunicazioni (chiamate di operazioni e trasmissioni di eventi) passanti attraverso i port. La linea fra l'istanza di **Amplif** e quella di **Tuning** rappresenta l'interazione fra queste due parti del componente.

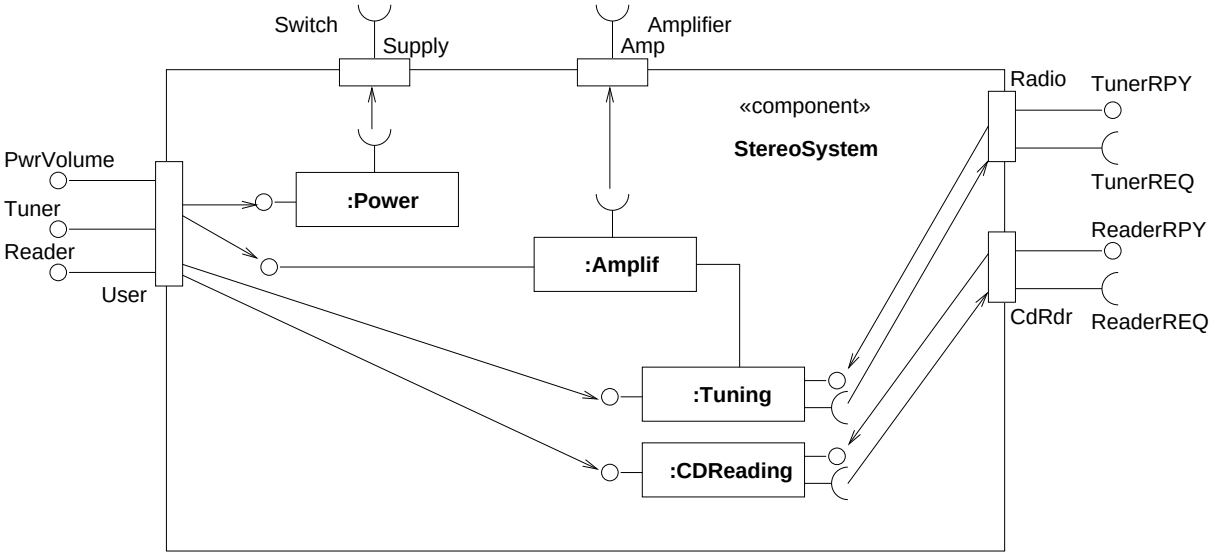


Figura 5.9: Componenti strutturati in UML.

La fig. 5.10 mostra il componente nel contesto del sistema complessivo.

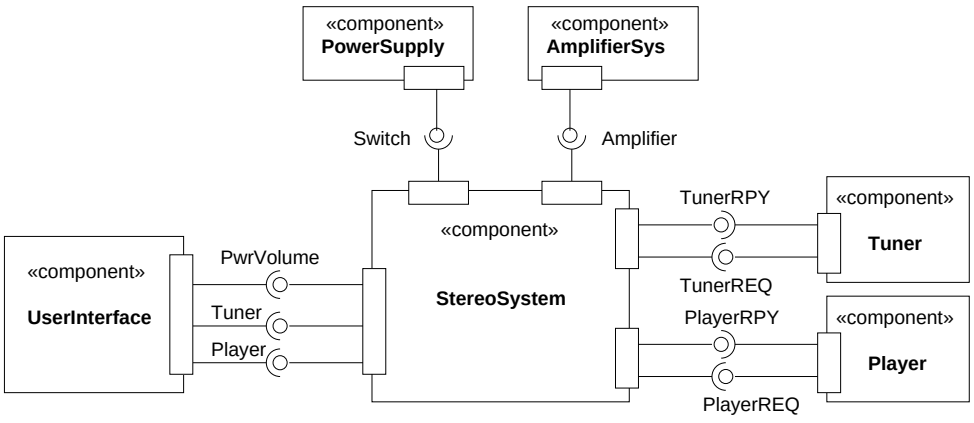


Figura 5.10: Un sistema di componenti.

5.4.2 Moduli generici

È molto comune che un algoritmo si possa applicare a tipi di dato diversi (per esempio, un algoritmo di ordinamento si può applicare a una sequenza di numeri interi, o di reali, o di stringhe), oppure che diversi tipi di dati, ottenuti per composizione di tipi piú semplici, abbiano la stessa struttura (per esempio, liste di interi, di reali, o di stringhe). I moduli generici permettono di "mettere a fattor comune" la struttura di algoritmi e tipi di dato, mettendo in evidenza la parte variabile che viene rappresentata dai parametri del modulo.

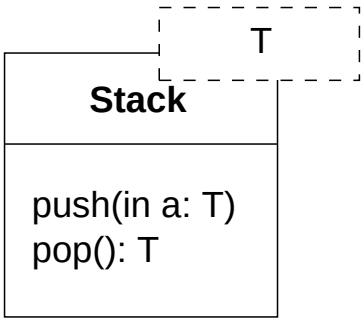


Figura 5.11: Classi generiche in UML (1).

Questo è utile dal punto di vista della comprensibilità del progetto, in quanto rende esplicito il fatto che certi componenti del sistema sono fatti allo stesso modo, e da quello della facilità di programmazione, in quanto permette di riusare facilmente dei componenti che altrimenti verrebbero riprogettati daccapo o adattati manualmente da quelli già esistenti, col rischio di introdurre errori. Inoltre l'uso di moduli generici non comporta una riduzione di efficienza rispetto all'uso di moduli specializzati, poiché i moduli generici vengono istanziati (cioè tradotti in moduli concreti, con valori definiti dei parametri) a tempo di compilazione invece che a tempo di esecuzione.

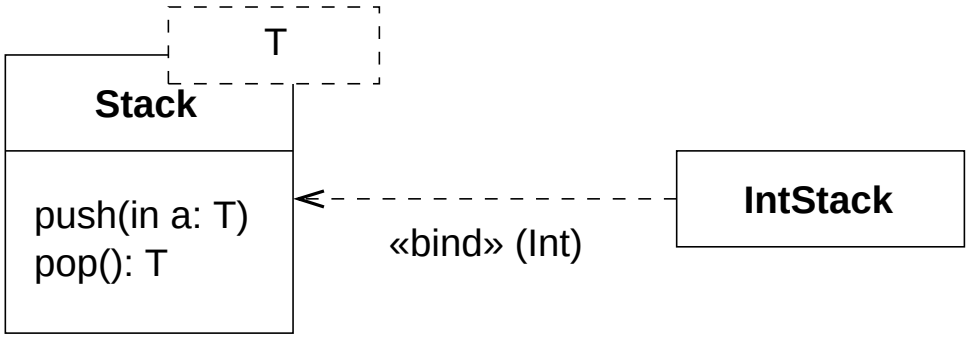


Figura 5.12: Classi generiche in UML (2).

In UML si possono rappresentare elementi di modello generici, e in particolare classi e package. La rappresentazione grafica di una classe generica è simile a quella di una classe non generica, con un rettangolo tratteggiato contenente i nomi ed eventualmente i tipi dei parametri, come in fig. 5.11. I parametri possono essere nomi di tipi (classi o tipi base), valori, operazioni, e si possono indicare dei valori di default.

La relazione fra una classe generica ed una sua istanza si può rappresentare esplicitamente con una dipendenza etichettata dallo stereotipo ≪bind≫ accompagnato dai valori dei parametri (fig. 5.12) o in modo implicito, in cui la classe istanza viene etichettata dal nome della classe generica seguito dai valori dei parametri (fig. 5.13).

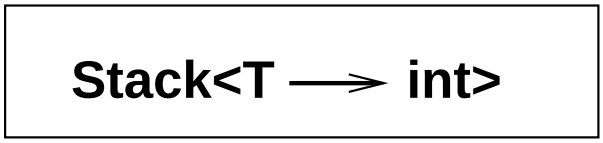


Figura 5.13: Classi generiche in UML (3).

Precisiamo che un'istanza di una classe generica è una classe, *non* un oggetto. Istanziare una classe generica e istanziare una classe sono due operazioni molto diverse, ma purtroppo nella terminologia corrente si usa la stessa parola.

5.4.3 Eccezioni

Come già accennato, la gestione degli errori e delle situazioni anomale è un aspetto della suddivisione di responsabilità fra moduli. I linguaggi che offrono il meccanismo delle *eccezioni* permettono di esprimere in modo chiaro e ben strutturato le soluzioni di questo problema. Il problema consiste nel decidere in quale modulo si deve trattare una data situazione eccezionale, dopodiché bisogna specificare in quali moduli si può verificare tale situazione, come viene riconosciuta, come viene segnalata, e infine come il controllo viene passato al modulo responsabile di gestire l'errore.

Nei linguaggi dotati di gestione delle eccezioni, i moduli in cui si può verificare un'eccezione contengono delle istruzioni che *sollevano* (*raise*), cioè segnalano tale eccezione. I moduli preposti a gestire l'eccezione contengono dei sottoprogrammi, detti *gestori* o *handler*, esplicitamente designati a tale scopo. A tempo di esecuzione, quando in un modulo si scopre il verificarsi di una situazione eccezionale (cosa che richiede dei controlli espliciti, tipicamente con istruzioni condizionali), l'istruzione che solleva l'eccezione causa l'interruzione dell'esecuzione del modulo ed il trasferimento del controllo al gestore piú vicino nella catena di chiamate che ha portato all'invocazione del modulo che ha sollevato l'eccezione.

Per fissare le idee, supponiamo che un modulo `A` chiami una funzione definita in un modulo `B`, che a sua volta chiama una funzione di un modulo `C`. Supponiamo inoltre che nell'esecuzione di `C` si possa verificare un'eccezione, e che nel modulo `A` (ma non in `B`) ci sia un gestore per tale eccezione. Se, quando si esegue `C`, l'eccezione viene sollevata, allora vengono interrotte le esecuzioni di `C` e `B`, e il controllo passa al gestore appartenente ad `A`. In questo gestore, se necessario, si può risollevare l'eccezione, delegandone la gestione ad altri moduli ancora, di livello piú alto. Questo può accadere se nel modulo `A` si scopre che l'eccezione non può essere gestita, oppure se `A` può eseguire solo una parte delle azioni richieste.

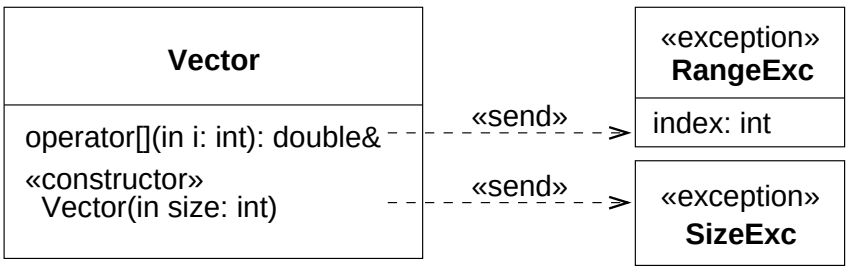


Figura 5.14: Eccezioni in UML1.

In UML1 le eccezioni si modellavano come oggetti che possono essere spediti (come segnali) da un oggetto all'altro. In questo modo, le classi che rappresentano eccezioni hanno lo stereotipo ≪exception≫. La dipendenza stereotipata ≪send≫ associa le eccezioni alle operazioni che le possono sollevare, come in fig. 5.14.

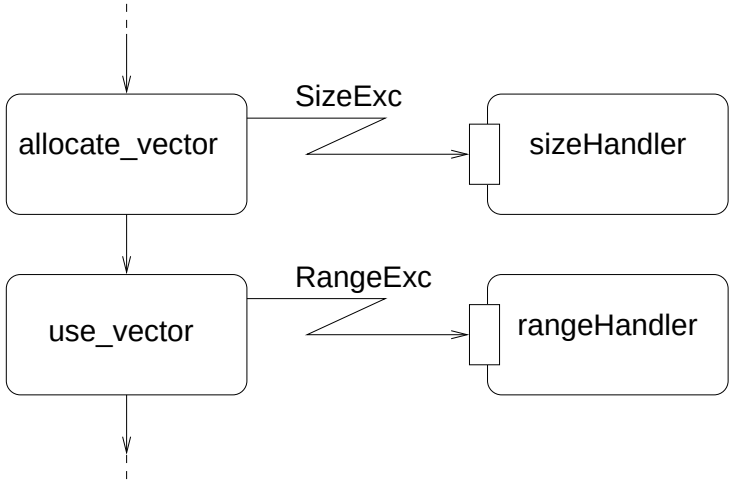


Figura 5.15: Eccezioni in UML2.

In UML2 le eccezioni non sono viste come segnali, ma come oggetti che vengono creati quando l'esecuzione di un'azione incontra una situazione anomala e vengono passati come parametri d'ingresso ad un'altra azione, il gestore di eccezioni. Questo meccanismo viene modellato nel diagramma di attività, con apposite notazioni che non tratteremo, limitandoci a mostrare, come esempio, un frammento di un possibile diagramma di attività (fig. 5.15). Nel diagramma delle classi, le eccezioni che si possono sollevare nel corso di una operazione possono essere elencate come valori della proprietà exceptions associata all'operazione. Notare le frecce a forma di fulmine (a ciel sereno?) che rappresentano il passaggio di controllo ai gestori di eccezioni.

Letture

Obbligatorie: cap. 3 e 4 Ghezzi, Jazayeri, Mandrioli.

Capitolo 6

Progetto orientato agli oggetti

Come abbiamo già osservato parlando della fase di analisi e specifica dei requisiti, nelle metodologie orientate agli oggetti un sistema viene visto come un insieme di oggetti interagenti e legati fra loro da vari tipi di relazioni. In fase di analisi e specifica dei requisiti si costruisce un modello in cui il sistema software (cioè il prodotto che vogliamo realizzare) viene visto nel contesto dell'ambiente in cui deve operare. Questo modello, in cui le entità appartenenti al dominio dell'applicazione (detto anche *spazio del problema*) sono preponderanti rispetto al sistema software, è il punto di partenza per definire l'architettura software, la cui struttura, negli stadi iniziali della progettazione, ricalca quella del modello di analisi.

La fase di progetto parte quindi dalle classi e relazioni definite in fase di analisi, a cui si aggiungono classi definite in fase di progetto di sistema e relative al dominio dell'implementazione (o *spazio della soluzione*). Nelle fasi successive del progetto questa struttura di base viene rielaborata, riorganizzando le classi e le associazioni, introducendo nuove classi, e definendone le interfacce e gli aspetti piú importanti delle implementazioni.

6.1 Un esempio

Immaginiamo di progettare un sistema per la gestione di una biblioteca. La fig. 6.1 dà un'idea estremamente semplificata di un possibile modello d'analisi. Ci sono due diagrammi di casi d'uso, di cui il secondo tiene conto del requisito che solo il bibliotecario interagisca direttamente col sistema. Si è mantenuta la versione originale del diagramma, che mostra il ruolo dell'utente, per ricordare che i servizi della biblioteca sono destinati all'utente, e lasciare spazio a future estensioni in cui l'utente potrebbe accedere al sistema direttamente. La dipendenza ≪trace≫ si usa per rappresentare la relazione storica fra diverse versioni di un modello o parti di esso.

La fig. 6.2 mostra un primo abbozzo del modello di progetto. È stata introdotta una classe (che in questo stadio sta per un intero sottosistema) che rappresenta l'interfaccia presentata dal sistema al bibliotecario. Questa classe ha lo stereotipo ≪boundary≫ ap-

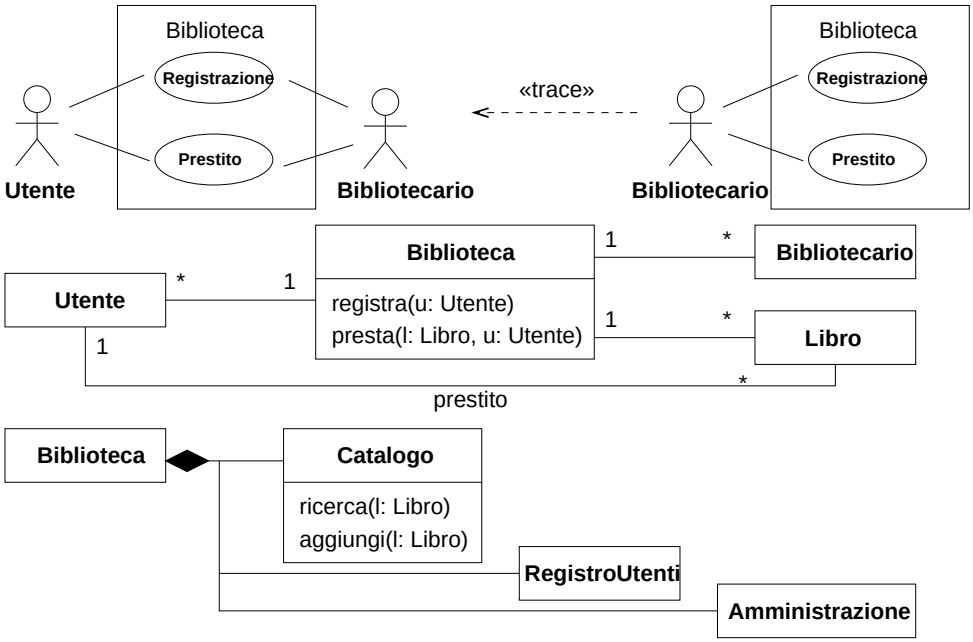


Figura 6.1: Un progetto OO (1).

punto per mostrare questo suo ruolo. È stato introdotto conseguentemente un registro dei bibliotecari. Le classi **Libro**, **Utente** e **Bibliotecario** hanno lo stereotipo ≪entity≫, comunemente usato per le classi destinate a contenere dati persistenti.

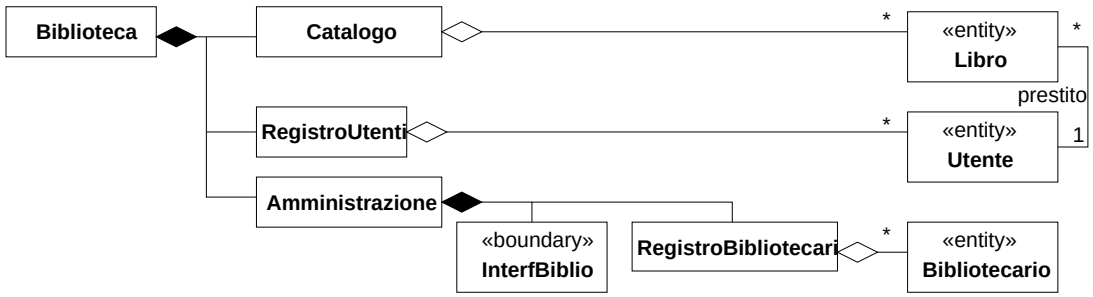


Figura 6.2: Un progetto OO (2).

La fig. 6.3 mostra un'ulteriore evoluzione del modello, in cui si implementa l'associazione logica prestito per mezzo di una classe apposita e di un registro dei prestiti incluso nel sottosistema di amministrazione della biblioteca. Inoltre si è scelto di implementare le varie aggregazioni per mezzo di classi istanziate dal template **list**, e si è aggiunto un sottosistema di persistenza, che deve provvedere a memorizzare i vari cataloghi e registri in un database.

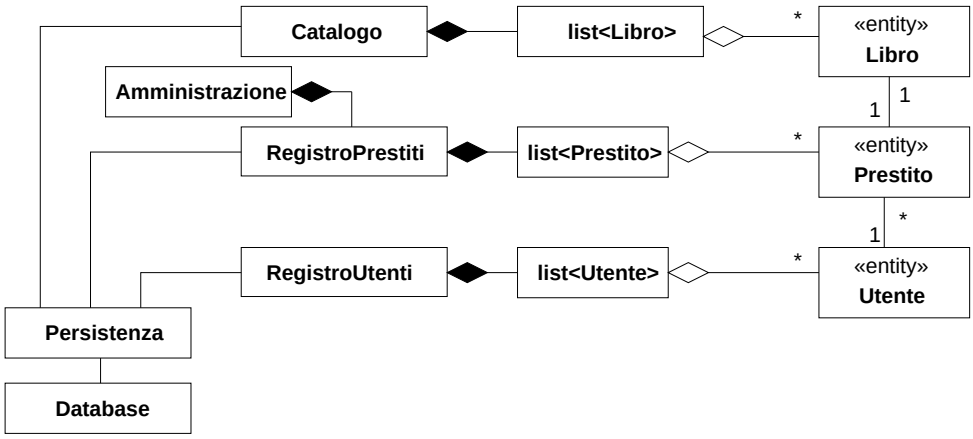


Figura 6.3: Un progetto OO (3).

6.2 Eredità e polimorfismo

In questa sezione tratteremo tre concetti che sono alla base della progettazione orientata agli oggetti: l'*eredità*, il *polimorfismo* e il *binding dinamico*.

6.2.1 Eredità

Nei linguaggi di programmazione, l'eredità è il meccanismo che inserisce gli attributi e operazioni di una classe base nelle classi derivate. Questo meccanismo si può sfruttare per tre scopi: (i) riprodurre nell'architettura software le relazioni di generalizzazione presenti nel dominio dell'applicazione, (ii) riusare moduli preesistenti, e (iii) implementare le relazioni di realizzazione.

Nella fase di analisi dei requisiti, l'eredità permette di modellare la relazione di generalizzazione (e quindi di specializzazione) fra entità del dominio di applicazione. Di solito, alle entità del dominio dell'applicazione devono corrispondere delle entità del sistema software. Nella fase di progetto viene definita un'architettura software in cui vengono mantenute le relazioni fra entità del dominio di applicazione, fra cui la generalizzazione, come mostra il seguente esempio (in C++):

```
class Person {
    char* name;
    char* birthdate;
public:
    char* getName();
    char* getBirthdate();
};

class Student : public Person {
    char* student_number;
public:
    char* getStudentNumber();
};
```

In questo caso il fatto che la classe `Student` erediti dalla classe `Person` corrisponde al fatto che nel dominio dell'applicazione (per esempio, il database dei dipendenti e degli studenti di una scuola) gli studenti sono un sottoinsieme delle persone. La classe derivata ha gli attributi e le operazioni della classe base, a cui aggiunge ulteriori attributi e operazioni.

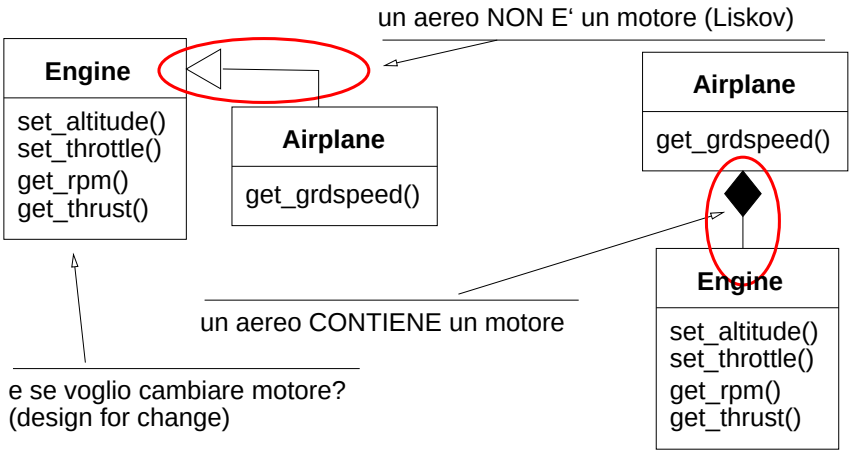


Figura 6.4: Riuso per eredità e per composizione.

In fase di progetto si può usare l'eredità come tecnica di riuso, cioè per riusare le operazioni di una classe base nel contesto di una classe derivata: per esempio, se si dispone di una classe **Shape** con le operazioni per disegnare e spostare una figura geometrica, si può definire una classe derivata **ColoredShape** che aggiunge le operazioni per colorare una figura, e riusa le operazioni della classe base per disegnarla e spostarla. Questo è conveniente quando la classe derivata ha una semantica (cioè uno scopo e un significato) analoga a quella della classe base, rispettando cosí il principio della Liskov. In molti casi, però, il meccanismo della composizione (usare un'istanza di una classe, o un puntatore ad essa, come attributo di un'altra) è piú flessibile dell'eredità. È comunque da evitare l'uso dell'eredità per costruire una classe derivata che non abbia una parentela logica con la classe base. Per esempio, sia le aziende che le persone hanno un nome e un indirizzo, ma questa non è una buona ragione per definire una classe **Azienda** come derivata di una classe **Persona**. Nella fig. 6.4, le operazioni per il controllo o la simulazione di un motore vengono ereditate, nel diagramma a sinistra, da una classe che rappresenta un aereo. Questo è sbagliato semanticamente perché gli aerei non sono un sottoinsieme dei motori, e anche pragmaticamente, perché lega strettamente l'implementazione della classe **Airplane** a quella della classe **Engine**. Il diagramma a destra mostra una soluzione appropriata, in cui il motore viene modellato correttamente come parte dell'aereo.

L'uso dell'eredità per implementare la relazione di realizzazione verrà discusso nella sezione seguente, dedicata al polimorfismo.

6.2.2 Polimorfismo e binding dinamico

Il polimorfismo ed il binding dinamico sono due concetti che nei linguaggi orientati agli oggetti sono strettamente legati fra di loro e con il concetto di eredità.

Il *polimorfismo* è la possibilità che un riferimento (per esempio un identificatore o un puntatore) denoti oggetti o funzioni di tipo diverso. Esistono diversi tipi di polimorfismo, e la forma di polimorfismo tipica dei linguaggi object oriented è quella basata sull'eredità: se una classe D deriva da una classe B, allora ogni istanza di D è anche un'istanza di B, per cui qualsiasi riferimento alla classe D è anche un riferimento alla classe B. Questo tipo di polimorfismo si chiama *polimorfismo per inclusione*. In C++, per esempio, un oggetto di classe D può essere assegnato ad un oggetto di classe B ed un valore di tipo "puntatore a D" o "riferimento a D" può essere assegnato, rispettivamente, ad una variabile di tipo "puntatore a B" o "riferimento a B". Osserviamo che in questo e in altri linguaggi esistono altre forme di polimorfismo, come l'overloading, che non sono legate all'eredità.

Il *binding* è il legame fra un identificatore (in particolare un identificatore di funzione) ed il proprio valore. Si ha un *binding* dinamico quando il significato di una chiamata di funzione (cioè il codice eseguito dalla chiamata) è noto solo a tempo di esecuzione: il binding dinamico è quindi il meccanismo che rende possibile il polimorfismo. In C++, le funzioni che vengono invocate con questo meccanismo sono chiamate *virtuali*. Se si chiama una funzione virtuale di un oggetto attraverso un puntatore, questa chiamata è polimorfica (per inclusione). Consideriamo questo (classico) esempio:

```
// file Shape.h
class Shape {
    Point position;
public:
    virtual void draw() {};
    virtual void move(Point p);
};

void
Shape::
move(Point p) { position = p; }

// file Circle.h
#include "Shape.h"

class Circle : public Shape {
    //...
public:
    void draw();
};

void
Circle::
draw() { cout << "drawing Circle\n"; };

// file Square.h
#include "Shape.h"

class Square : public Shape {
```

```
    //...
public:
    void draw();
};

void
Square::
draw() { cout << "drawing Square\n"; };

// file main.cc
#include "Circle.h"
#include "Square.h"

void
drawall(Shape** shps)
{
    for (int i = 0; i < 2; i++)
        shps[i]->draw();
}

main()
{
    Shape* shapes[2];
    shapes[0] = new Circle;
    shapes[1] = new Square;
    drawall(shapes);
}
```

La struttura del programma può essere rappresentata in UML come in fig. 6.5, dove lo stereotipo ≪utility≫ índica che una classe contiene solo operazioni o dati globali (è quindi un espediente per rappresentare moduli non riconducibili al modello object-oriented).

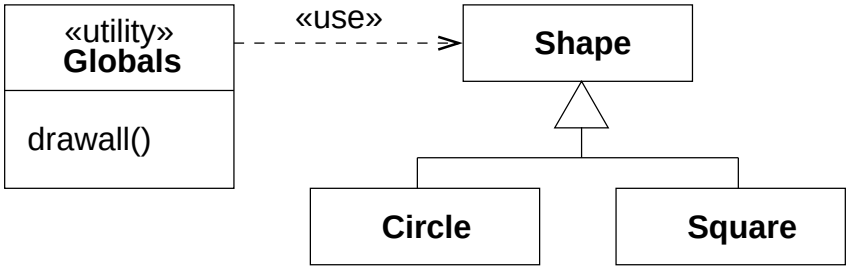


Figura 6.5: Esempio: polimorfismo (1).

Nella funzione `drawall()` il tipo dell'oggetto a cui viene applicata la funzione `draw()` (che deve disegnare una figura sullo schermo) è noto soltanto a tempo di esecuzione: si sa solo che l'oggetto apparterrà alla classe `Shape` o a una classe derivata da questa. La funzione `draw()` è quindi polimorfica, e viene chiamata con binding dinamico. Il fatto che la funzione `drawall()` ignori il tipo degli oggetti su cui deve operare ne migliora grandemente la modularità e la riusabilità rispetto ad un'implementazione che invece richieda una conoscenza statica dei tipi. Questa funzione è completamente disaccoppiata dall'implementazione delle classi `Circle` e `Square`, e inoltre continua a funzionare, immutata, anche se si aggiungono altre classi derivate da `Shape`.

6.2.3 Classi astratte e interfacce

Osserviamo che nel nostro esempio l'operazione `draw()` della classe `Shape` ha un'implementazione banale, è un'operazione nulla, poiché non si può disegnare una forma generica: il concetto di "forma" (*shape*) è astratto, e si possono disegnare effettivamente solo le sue realizzazioni concrete, come "cerchio" e "quadrato". La classe `Shape` è in realtà un cattivo esempio di programmazione, poiché non rappresenta adeguatamente il concetto reale che dovrebbe modellare, e questa inadeguatezza porta alla realizzazione di software poco affidabile. Infatti è possibile istanziare (contro la logica dell'applicazione) un oggetto `Shape` a cui si potrebbe applicare l'operazione `draw()`, ottenendo un risultato inconsistente.

Una prima correzione a questo errore di progetto consiste nel rendere esplicito il fatto che la classe rappresenta un concetto astratto. In C++, questo si ottiene specificando che `draw` è un'operazione virtuale pura, per mezzo dello specificatore '`= 0`':

```
// file Shape.h
class Shape {
    Point position;
public:
    virtual void draw() = 0;
    virtual void move(Point p);
    //...
};

void
Shape::
move(Point p) { position = p; }
```

La classe Shape è ora una classe astratta, cioè non istanziabile a causa dell'incompletezza della sua implementazione (fig. 6.6).

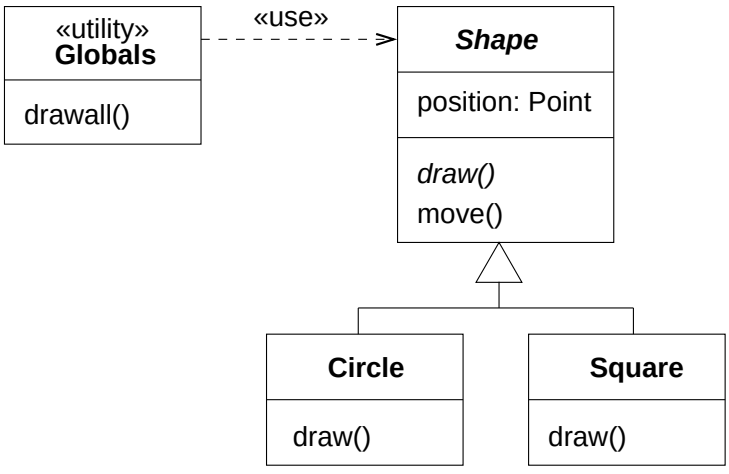


Figura 6.6: Esempio: polimorfismo (2).

Una struttura ancor piú modulare si può ottenere rappresentando esplicitamente l'interfaccia, separandola dall'implementazione:

```
// file Shape.h
class IShape {
public:
    virtual void draw() = 0;
    virtual void move(Point p) = 0;
};

class Shape : public IShape {
    Point position;
public:
    virtual void draw() = 0;
    virtual void move(Point p);
};

void
Shape::
move(Point p) { position = p; }

class Circle : public Shape {
    //...
public:
    void draw();
};

void
Circle::
draw() { cout << "drawing Circle\n"; }

class Square : public Shape {
    //...
public:
    void draw();
};

void
Square::
draw() { cout << "drawing Square\n"; }

void
drawall(IShape** shps)
{
    for (int i = 0; i < 2; i++)
        shps[i]->draw();
}

main()
{
    IShape* shapes[2];
    shapes[0] = new Circle;
    shapes[1] = new Square;
    drawall(shapes);
}
```

In questa versione la classe `IShape` definisce l'interfaccia comune a tutti gli oggetti che possono essere disegnati o spostati, la classe `Shape` definisce la parte comune delle loro implementazioni (il fatto di avere una posizione e un'operazione che modifica tale

posizione), e le classi rimanenti definiscono concretamente i metodi per disegnare le varie figure. Questa struttura si può schematizzare come in fig. 6.7.

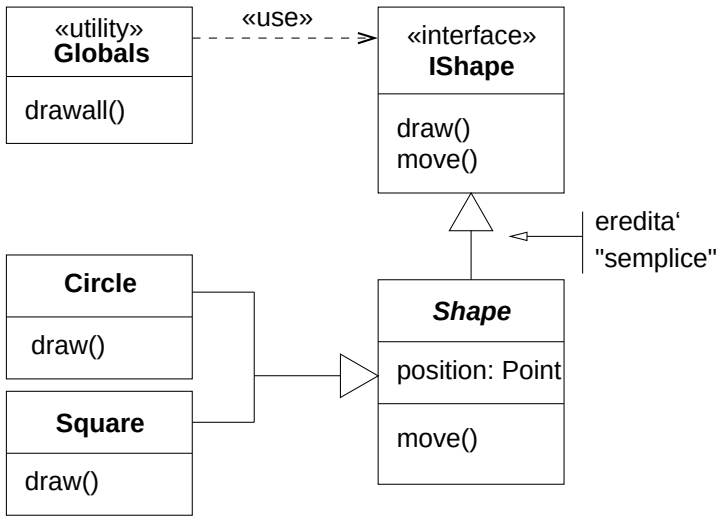


Figura 6.7: Esempio: polimorfismo (3).

6.2.4 Eredità multipla

L'eredità multipla si ha quando una classe ha piú di una classe base diretta. Questa possibilità è particolarmente utile per ottenere un'interfaccia come composizione di altre interfacce. Le interfacce delle classi base rappresentano diversi aspetti delle entità modellate dalla classe derivata. I clienti della classe derivata possono usare indipendentemente questi diversi aspetti, ottenendo cosí un disaccoppiamento fra le varie classi migliore di quello che si avrebbe riunendoli in un'interfaccia unica.

Nel seguente esempio (in Java) un videogioco deve simulare dei velivoli. Ciascuno di questi viene visto sotto due aspetti: la simulazione del comportamento (`Aircraft`) e la raffigurazione grafica (`Drawable`). Questi due aspetti vengono gestiti da due sottosistemi distinti, rispettivamente `AirTrafficCtrl` e `DisplayMgr`.

```
abstract class Aircraft {
    public double speed;
    public abstract void fly(AirTrafficCtrl atc);
}

interface Drawable {
    void draw();
}

class JetLiner extends Aircraft implements Drawable {
    public JetLiner() { speed = 850.0; }
    public void fly(AirTrafficCtrl atc) { /* flight simulation */ }
    public void draw() { /* graphic rendering */ }
}
```

```
class DisplayMgr {
    private Drawable[] d;
    public DisplayMgr(Drawable[] dd) { d = dd; }
    public void display() { /* use draw() */ }
}

class AirTrafficCtrl {
    private Aircraft c[];
    private DisplayMgr m;
    public AirTrafficCtrl(Aircraft[] ac, DisplayMgr dm)
       { c = ac; m = dm; }
    public void simulate() { /* use fly() */ }
}

public class VideoGame {
    public static void main(String[] args)
    {
        JetLiner planes[] = new JetLiner[2];
        planes[0] = new JetLiner();
        planes[1] = new JetLiner();
        DisplayMgr dm = new DisplayMgr(planes);
        AirTrafficCtrl atc = new AirTrafficCtrl(planes, dm);
        dm.display();
        atc.simulate();
    }
}
```

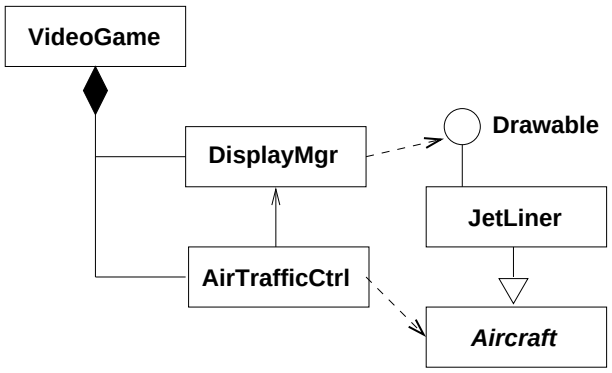


Figura 6.8: Esempio: polimorfismo (4).

In Java un `interface` corrisponde ad una classe virtuale pura del C++. Le parole chiave `extends` e `implements` denotano l'eredità rispettivamente da classi ordinarie (eventualmente astratte) e da classi virtuali pure. Il Java permette di ereditare direttamente da una sola classe ordinaria e da piú classi virtuali pure.

La fig. 6.8 è una rappresentazione diretta del codice. Una rappresentazione piú strutturata è in fig. 6.9, in cui **Aircraft** è un'interfaccia (non piú una classe astratta) e **JetLiner** è un componente strutturato, contenente i sottosistemi **Rendering** e **Simulation**.

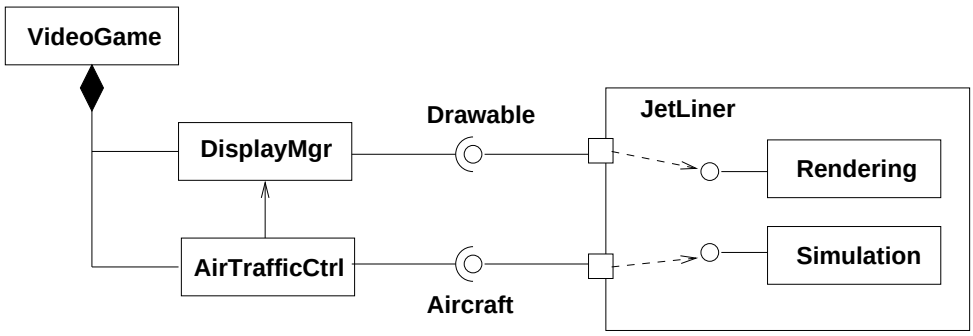


Figura 6.9: Esempio: polimorfismo (5).

6.3 Progetto di sistema

In questa sezione consideriamo alcune linee guida per il progetto di sistema, nel quale si fanno delle scelte di carattere fondamentale che indirizzano le attività successive. Le scelte principali riguardano la struttura generale del sistema, di cui si individuano i componenti principali ed i meccanismi di interazione.

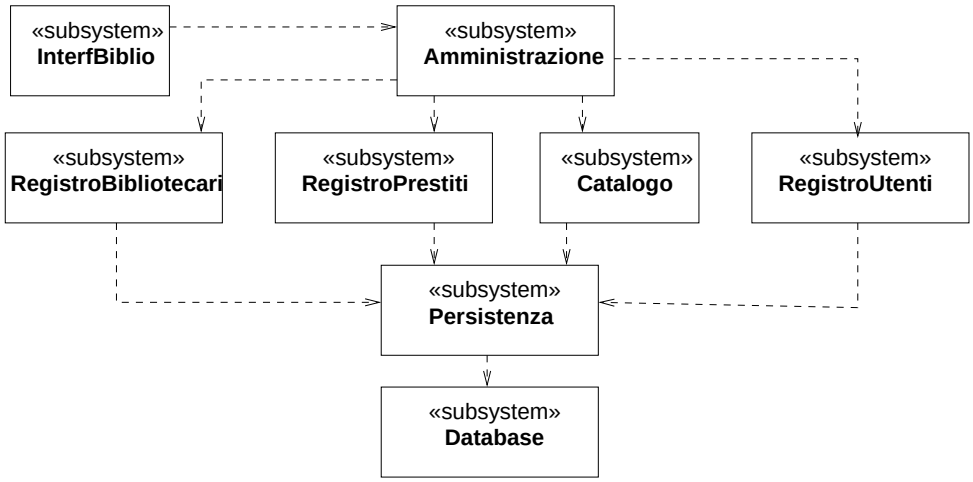


Figura 6.10: Esempio: progetto di sistema (1).

Altre scelte riguardano la gestione delle condizioni al contorno: bisogna cioè stabilire il comportamento del sistema nelle situazioni di inizializzazione, terminazione, ed errore. Bisogna anche stabilire delle priorità fra obiettivi contrastanti (per esempio, velocità di esecuzione e risparmio di memoria), in modo da risolvere conflitti fra soluzioni alternative che si possono presentare nel corso del progetto. Queste priorità, però, non dovranno essere applicate meccanicamente: per esempio, l'aver stabilito che la velocità di esecuzione è prioritaria rispetto al risparmio di memoria non giustifica un sovradimensionamento della memoria, se questo ha un costo eccessivo rispetto al guadagno di velocità ottenibile.

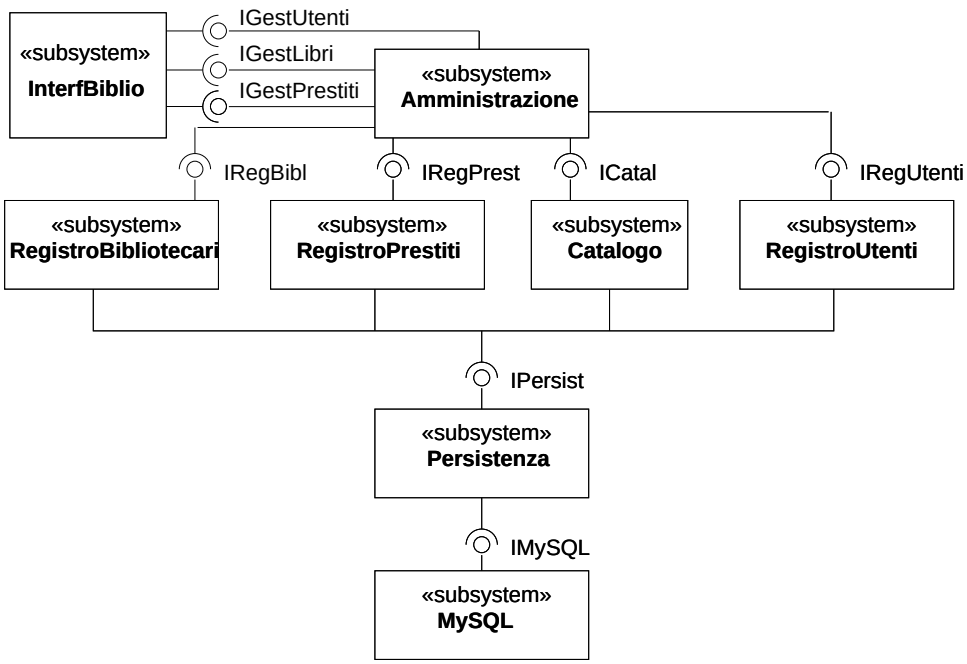


Figura 6.11: Esempio: progetto di sistema (2).

6.3.1 Ripartizione in sottosistemi

Il primo passo nel progetto di sistema consiste nell'individuare i sottosistemi principali. Riprendendo l'esempio del software per la gestione di una biblioteca, la fig. 6.10 mostra la struttura di tale software a livello di sottosistemi.

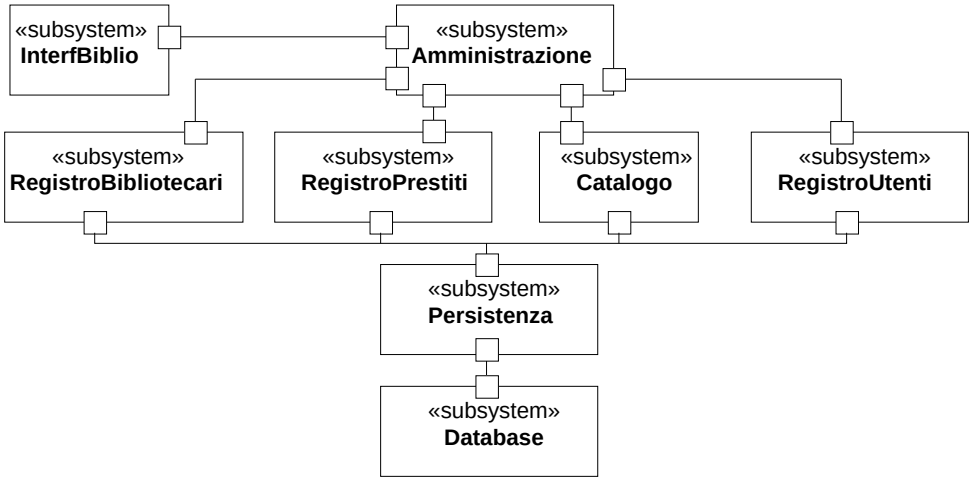


Figura 6.12: Esempio: progetto di sistema (3).

In questo esempio si sono mostrate solo le relazioni di dipendenza fra i moduli. Un passo successivo consiste nell'individuare le interfacce, mostrate nel diagramma di fig. 6.11, in cui si vede anche che si è scelto di implementare il database con un prodotto software ester-

no, MySQL. La fig. 6.12 mostra un'altra versione, in cui si tralascia la rappresentazione esplicita delle interfacce.

Architetture standard

Esistono alcuni schemi di interconnessione (o *topologie*) fra sottosistemi che, insieme a determinati schemi di interazione, vengono usati comunemente e spesso sono caratteristici di certi tipi di applicazioni. Di solito conviene scegliere uno di questi schemi come punto di partenza del progetto.

Esempi di tali schemi sono le architetture a *pipeline*, *client-server* a uno o due livelli, a *repository*, mostrate in fig. 6.13. Alcune applicazioni che usano tali architetture sono, rispettivamente, i compilatori, i servizi web piú semplici, i servizi web per l'accesso a basi di dati, gli strumenti CASE. Osserviamo che il modello client-server ed il modello repository hanno la stessa topologia, ma una diversa interazione fra i sottosistemi: nel modello client-server i clienti sono indipendenti, mentre nel modello repository i vari componenti accedono al repository per scambiare dati e collaborare.

Altre categorie di applicazioni per cui si tende a usare delle architetture standard sono i sistemi interattivi, i simulatori, i sistemi real-time, i sistemi a transazioni. Quando s'intraprende il progetto di un nuovo sistema conviene verificare se esso appartiene ad una delle categorie note, e prendere a modello lo schema generale di architettura tipico di tale categoria.

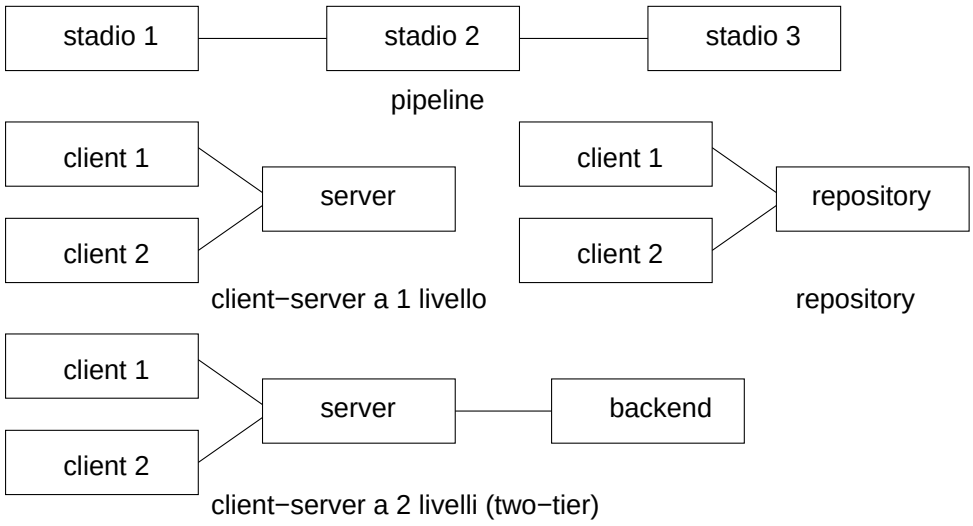


Figura 6.13: Alcune architetture standard.

Scomposizione in strati e partizioni

Un metodo di applicabilità generale per organizzare un'architettura si basa sulla scomposizione per *strati* e per *partizioni*. Nella scomposizioni in strati ogni strato è un sottosistema

che offre dei servizi ai sottosistemi di livello superiore e li implementa attraverso i servizi offerti dai sottosistemi a livello inferiore. Nella scomposizione in partizioni ogni partizione è un sottosistema che realizza una funzione del sistema complessivo. I due criteri generalmente vengono applicati insieme, poiché una partizione può essere stratificata ed uno strato può essere diviso in partizioni.

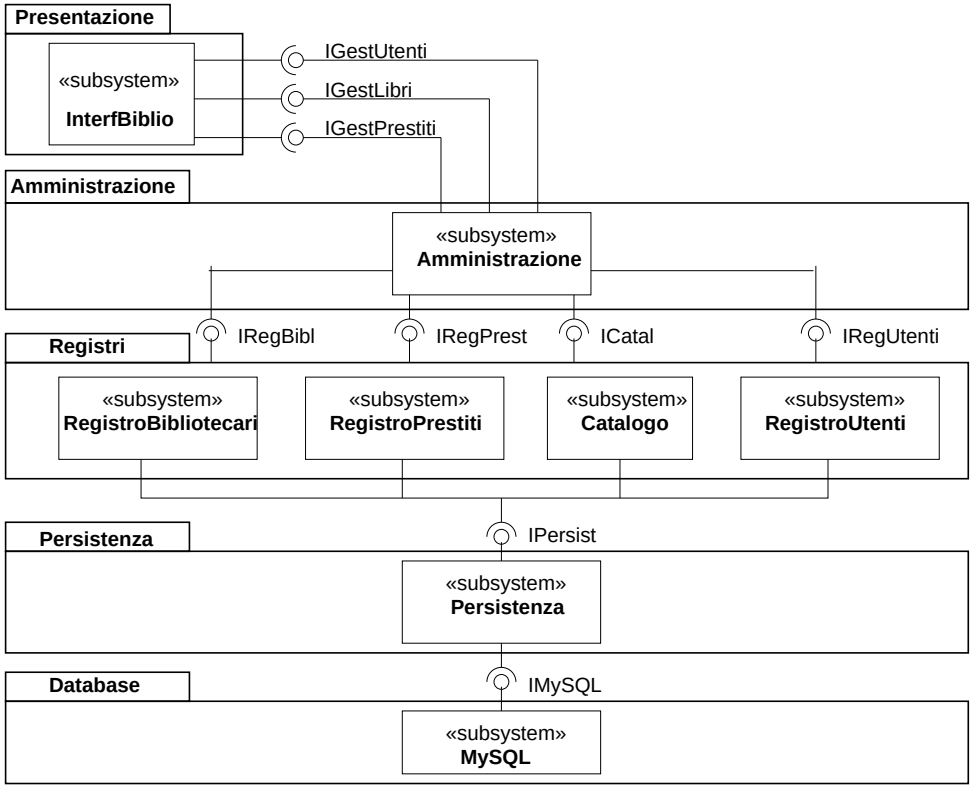


Figura 6.14: Scomposizione per strati e partizioni (1).

L'individuazione delle partizioni, cioè dei sottosistemi responsabili di realizzare le diverse funzioni dell'applicazione, viene guidata principalmente dalle informazioni raccolte nei documenti di specifica dei requisiti, per esempio dai diagrammi dei casi d'uso.

Gli strati vengono individuati in base ai diversi livelli di astrazione dei servizi richiesti per realizzare il sistema: esempi tipici di architetture a strati sono i sistemi operativi ed i protocolli di comunicazione.

Un esempio di scomposizione per strati è mostrato in fig. 6.14, dove lo schema a componenti dell'esempio sul software per una biblioteca viene ridisegnato raggruppando i sottosistemi in cinque strati, rappresentati per mezzo di package. I quattro sottosistemi dello strato **Registri** sono partizioni dello stesso, e anch'esse avrebbero potuto essere messe in evidenza con dei package.

Un altro modo di rappresentare una struttura a strati e partizioni viene mostrato in fig. 6.15. Questo stile non fa parte del linguaggio UML, ma è molto diffuso, assieme alla variante in cui i vari strati sono disegnati come corone circolari concentriche.

Figura 6.15: Scomposizione per strati e partizioni (2).

6.3.2 Librerie e framework

Durante la scomposizione in sottosistemi bisogna infine considerare la disponibilità di librerie e di *framework*. Precisiamo che qui usiamo il termine "libreria" in un significato diverso da quello comune: di solito si intende con questo termine un componente fisico, cioè un file contenente dei moduli collegabili (p.es, i file `.a` e `.so` sui sistemi Unix, `.lib` e `.dll` sui sistemi Windows), o analoghi moduli precompilati per linguaggi interpretati (come i file `.jar` in ambiente Java). Qui invece parliamo di librerie e framework dal punto di vista logico, ricordando che tutti e due i tipi di componenti vengono realizzati e distribuiti come librerie fisiche.

Una libreria logica è una raccolta di componenti che offrono servizi ad un livello di astrazione piuttosto basso. Un framework contiene invece dei componenti ad alto livello di astrazione che offrono uno schema di soluzione preconfezionato per un determinato tipo di problema. Le librerie, quindi, si usano "dal basso verso l'alto", assemblando componenti semplici e predefiniti per ottenere strutture complesse specializzate, mentre i framework si usano "dall'alto verso il basso", riempiendo delle strutture complesse predefinite (delle "intelaiature") con dei componenti semplici specializzati.

Le librerie e i framework sono componenti pronti all'uso (o, come si dice, *off the shelf*), generalmente non modificabili e forniti da produttori esterni. Nella maggior parte dei casi l'uso di tali componenti è molto vantaggioso dal punto di vista sia del processo di sviluppo che della qualità, e in particolare dell'affidabilità, del prodotto finale. Ovviamente una scelta accurata fra i componenti disponibili ha un'importanza cruciale per il successo del progetto, e questa scelta dipende sia da fattori tecnici, come le funzioni e prestazioni dei componenti, sia da fattori economici ed organizzativi, come le licenze d'uso e l'assistenza tecnica. Se fra i componenti pre-esistenti non se ne trovano di adatti allo scopo, naturalmente resta la possibilità di svilupparli *ex novo*.

Infine, conviene ricordare che una libreria o un framework possono essi stessi essere sviluppati e venduti come prodotto finale.

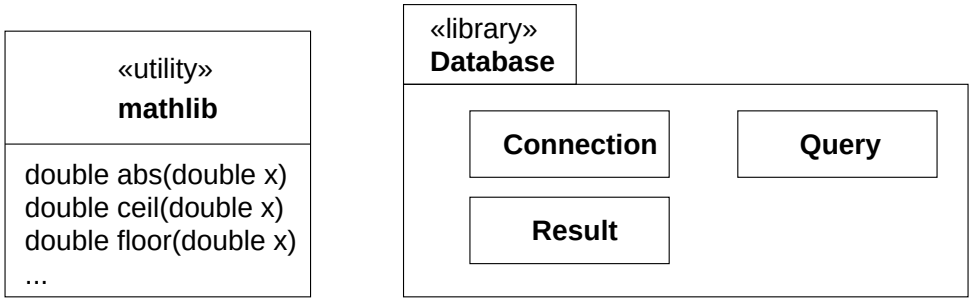


Figura 6.16: Due tipi di librerie.

Librerie

Una libreria può essere una raccolta di sottoprogrammi globali (cioè non appartenenti a classi) o di classi (fig. 6.16). In tutti e due i casi l'insieme delle operazioni disponibili viene chiamato *interfaccia di programmazione* o *API* (*Application Programming Interface*). Nel primo caso, la libreria si può rappresentare in UML come una classe, un componente o un package con lo stereotipo ≪utility≫. Nel progetto architetturale non è necessario mostrare la struttura interna della libreria, e di solito non serve nemmeno elencare le operazioni dell'interfaccia di programmazione. Nemmeno nel progetto in dettaglio è necessario, generalmente, mostrare queste informazioni, anche se in certi casi può essere utile esplicitare quali parti dell'interfaccia di programmazione vengono usate, e con quali classi della libreria si interagisce.

Framework

Come abbiamo detto, un framework offre uno schema di soluzione preconfezionato per un determinato tipo di problema, e viene usato in un'applicazione particolare specializzandone alcune caratteristiche. Questa specializzazione avviene generalmente per mezzo del polimorfismo: il framework offre al progettista delle classi e delle interfacce il cui comportamento viene specializzato derivando altre classi in cui si possono ridefinire i metodi delle classi e interfacce originarie, ed aggiungere ulteriori funzioni.

Un esempio (evidentemente semplificato) di framework viene mostrato in fig. 6.17. Un'interfaccia grafica è formata da elementi (**Widget**) che possono essere composti o semplici. Gli elementi composti sono, per esempio, finestre, pannelli e menú, e tutti hanno l'operazione add(), ereditata da **Compound**, che permette di aggiungere un widget all'elemento composto. Gli elementi semplici sono, per esempio, bottoni e campi per l'inserimento di testi, ed hanno varie operazioni specifiche di ciascuna classe. In particolare, la classe *Button* è astratta, avendo nella propria interfaccia l'operazione astratta action() che rappresenta, una volta implementata, l'azione da eseguire quando un particolare bottone viene "cliccato".

Supponiamo di voler progettare un'interfaccia grafica in cui una finestra contiene un bottone e un campo testo, dove il bottone serve a scrivere sull'uscita standard il conte-

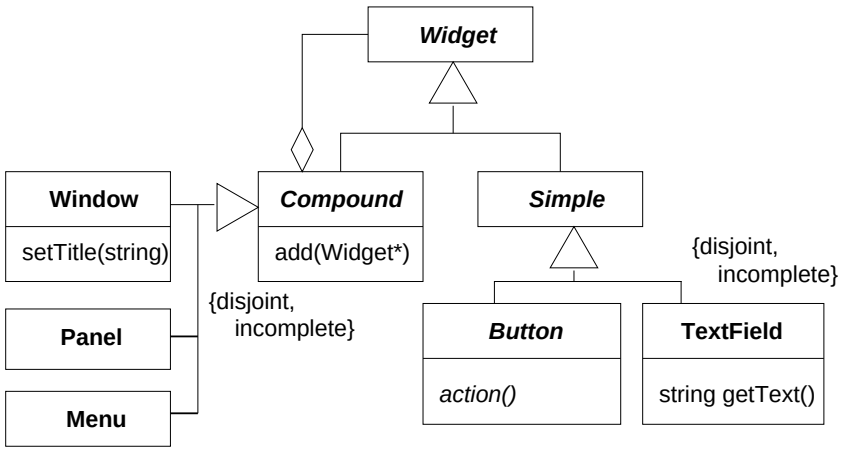


Figura 6.17: Un semplice framework.

nuto del campo testo preventivamente riempito dall'utente. Tutti i meccanismi a basso livello per la gestione del mouse, della tastiera, della visualizzazione grafica sono forniti dal framework, oltre alla struttura logica ed ai meccanismi di interazione fra i vari elementi. Il programmatore deve soltanto scrivere una classe (p.es., **MyButton**) derivata da **Button** che implementi l'operazione action() nel modo desiderato, quindi scrivere il programma principale dell'interfaccia grafica, in cui si istanzia una finestra e vi si aggiungono un'istanza di **TextField** e una di **MyButton**. La fig. 6.18 mostra la struttura risultante.

Un framework può essere rappresentato in UML come un componente o un package. Nemmeno per i framework, come già visto per le librerie, è generalmente richiesta una rappresentazione della struttura interna.

6.3.3 Sistemi concorrenti

Il comportamento dei sistemi complessi generalmente si può descrivere come un insieme di attività concorrenti. Il progettista deve individuare tali attività e stabilire quali sottosistemi le devono svolgere, quindi analizzare le possibili interazioni fra i sottosistemi, come scambi di messaggi ed accessi a risorse condivise, e le interazioni fra il sistema e l'ambiente esterno.

In questa sezione tratteremo sistemi concorrenti *non distribuiti*, cioè eseguiti su un singolo calcolatore.

Meccanismi di base per la concorrenza

Per trattare, anche se a livello introduttivo, il progetto di sistemi concorrenti, conviene richiamare alcuni concetti fondamentali, rimandando uno studio piú approfondito ad altri insegnamenti.

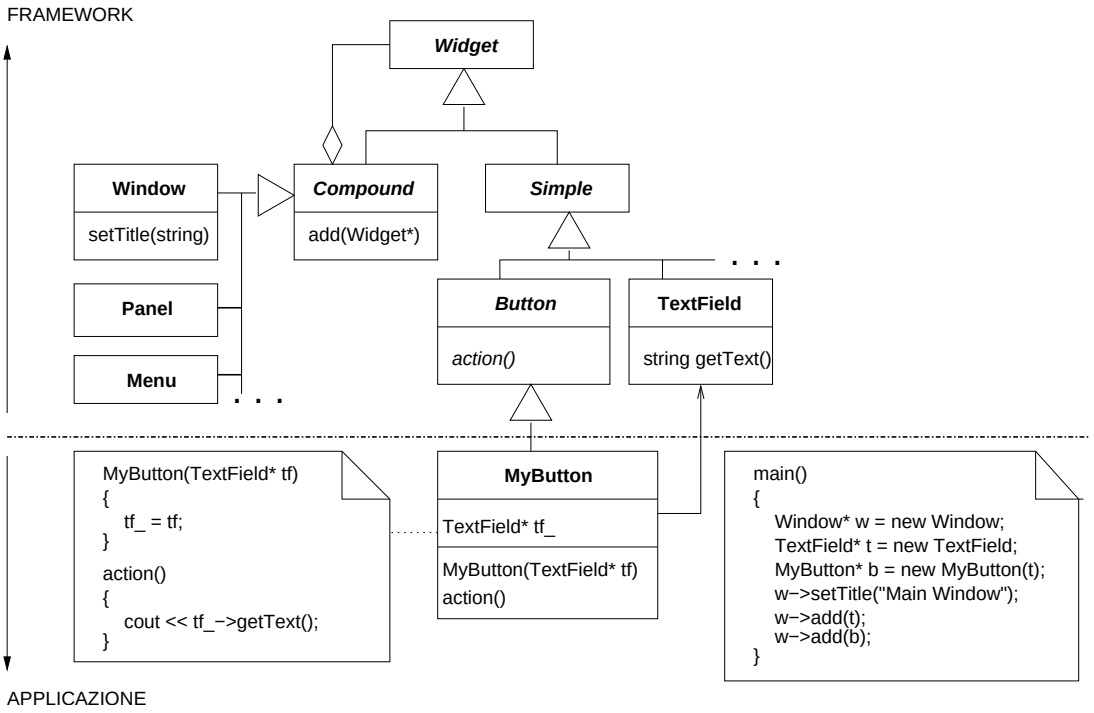


Figura 6.18: Uso di un framework in un'applicazione.

Lo stato di un programma sequenziale in qualsiasi passo delle sua esecuzione è definito dal valore del contatore di programma e dal contenuto dello stack, della memoria di lavoro (cioè l'insieme delle variabili definite nel programma), e delle strutture dati usate dal sistema operativo che si riferiscono al programma. L'insieme di queste informazioni, o piú precisamente la successione dei valori assunti da tali informazioni, insieme al codice macchina (detto *testo*) del programma, costituisce il *processo* associato ad un'istanza di esecuzione. Fra queste informazioni, il contatore di programma e lo stack definiscono il *flusso di controllo* (*thread of control*) del processo.

Il programmatore usa delle chiamate di sistema, come le primitive `fork()` ed `exec()` nei sistemi Unix, per creare i processi. Altre primitive (oppure chiamate di libreria a livello piú alto delle primitive di nucleo) permettono la *comunicazione* interprocesso e la *sincronizzazione* fra processi. Sempre riferendoci ai sistemi Unix, fra i meccanismi di comunicazione interprocesso citiamo i *pipe*, i *socket*, e la *memoria condivisa*; fra i meccanismi di sincronizzazione la chiamata `wait()` e i *semafori*. Usando questi strumenti, si può progettare un'applicazione concorrente composta da piú processi.

La maggior parte dei sistemi operativi, inoltre, permette a un processo di eseguire in modo concorrente piú flussi di controllo. Ciascun flusso di controllo viene realizzato da un'entità chiamata *thread* o *processo leggero* (*lightweight process*), definita da uno stack e un contatore di programma. In un processo con piú flussi di controllo, detto *multithreaded*, i thread condividono la memoria di lavoro e il codice eseguibile. In generale ogni thread può eseguire un proprio segmento di codice, ma tutti i segmenti di codice associati ai thread fanno parte del testo del processo a cui appartengono.

In pratica, il programmatore deve scrivere in un sottoprogramma le azioni che devono essere eseguite da un thread, e poi, usando apposite chiamate di sistema, chiedere al sistema operativo di creare il thread entro un processo. Anche per i thread ci sono delle chiamate di sistema per il controllo dell'esecuzione e per i meccanismi di sincronizzazione. Lo standard adottato dalla maggior parte dei sistemi Unix è l'interfaccia *Posix Threads*[1].

Oggetti attivi

Il flusso di controllo è un concetto fondamentale nell'analisi dei sistemi concorrenti. Nella sezione precedente è stato definito come la sequenza di valori assunti dal contatore di programma e dallo stack di un thread. In modo piú astratto, si può identificare con la sequenza di azioni svolte da un agente autonomo.

In un progetto orientato agli oggetti si definiscono *attivi* quegli oggetti che hanno la capacità di attivare un flusso di controllo. Il flusso di controllo di un oggetto attivo è costituito principalmente da chiamate di operazioni su oggetti (detti *passivi*) privi di un flusso di controllo indipendente, ma in un sistema concorrente il flusso di controllo di un oggetto attivo può interagire anche con quelli di altri oggetti attivi: un oggetto attivo può invocare operazioni di altri oggetti attivi (direttamente o inviando messaggi) o condividere oggetti passivi con altri oggetti attivi.

Per esempio, nel codice seguente ci sono due processi `p1` e `p2`, i cui programmi principali sono oggetti attivi (anche se, sintatticamente, non sono istanze di una classe). Le istanze `c1` e `c2` delle classi `class1` e `class2` sono oggetti passivi attraversati dal flusso di controllo dei rispettivi processi.

```
class class1 {                          | class class2 {
public:                                 | public:
    sub1();                             |     sub2();
};                                      | };
                                        |
void                                    | void
sub1()                                  | sub2()
{                                       | {
    // ...                              |     // ...
}                                       | }
                                        |
int                                     | int
main()     // p1                        | main()     // p2
{                                       | {
    if (fork() == 0) // proc. figlio    |     class2 c2;
        execlp("p2", NULL, NULL);       |     c2.sub2();
    else             // proc. padre     | }
        class1 c1;                      |
        c1.sub1();                      |
    }                                   |
}                                       |
```

[1] `http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html`

In UML gli oggetti attivi e le relative classi sono caratterizzati dalla proprietà isActive e vengono rappresentati con i lati verticali raddoppiati. I messaggi ed i segnali vengono rappresentati per mezzo di frecce nei diagrammi di sequenza e di comunicazione, ed è possibile usare notazioni diverse per esprimere diversi modi di sincronizzazione associati ai messaggi. Per esempio, si possono distinguere i messaggi *sincroni* (rappresentati da frecce a punta piena: $\longrightarrow\!\!\blacktriangleright$), in corrispondenza dei quali il mittente resta in attesa che il ricevente completi l'azione richiesta dal messaggio, e i messaggi *asincroni* (frecce a punta aperta: $\longrightarrow$), in cui il mittente continua la sua attività indipendentemente da quella del ricevente. Inoltre, le operazioni eseguite in modo concorrente possono essere specificate con la proprietà concurrent.

Per individuare i flussi di controllo indipendenti (almeno i piú importanti, in fase di progetto di sistema) bisogna analizzare le informazioni che si ricavano dal modello dinamico, che in un progetto orientato agli oggetti è rappresentato dai diagrammi di stato, di sequenza, di comunicazione e di attività (o da informazioni analoghe se si usano notazioni diverse dall'UML).

Esempio

La fig. 6.19 mostra, come sempre in modo semplificato, un modello di analisi per un sistema di controllo che deve rilevare due condizioni pericolose in un impianto: incendio o perdita di alimentazione elettrica (esempio adattato da [8]). Il diagramma delle classi mostra i sottosistemi principali: un controllore centrale, due sottosistemi di monitoraggio, rispettivamente per la temperatura e l'alimentazione, un certo numero di sensori per la temperatura e per la tensione di rete, e un sottosistema per la gestione delle situazioni di emergenza. Il diagramma di sequenza è rappresentativo di una serie di diagrammi che mostrano il comportamento del sistema in varie situazioni; questo particolare diagramma mostra una possibile sequenza in cui si rileva un incendio. Inoltre, dalle specifiche risulta che l'incendio è piú pericoloso della mancanza di alimentazione, per cui una eventuale attività di gestione di una mancanza di elettricità deve essere interrotta in caso di incendio[2].

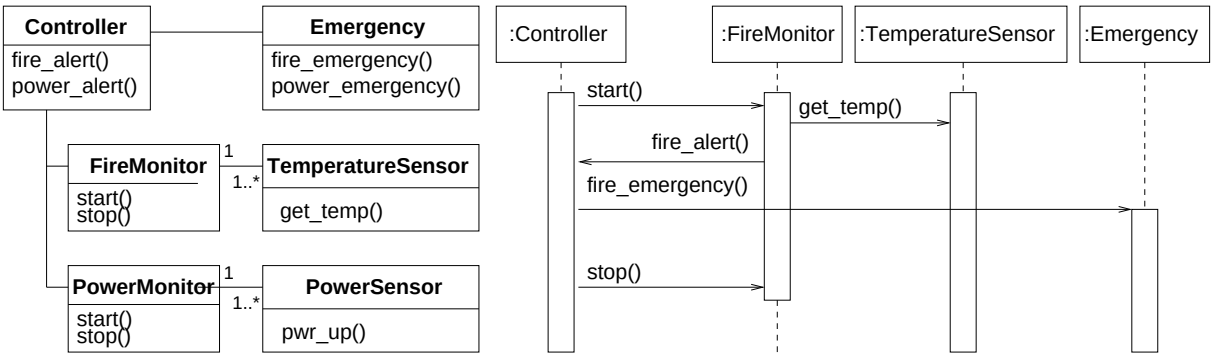


Figura 6.19: Un sistema concorrente (1).

[2]Si suppone che l'impianto antincendio abbia un'alimentazione autonoma ed affidabile.

Da quanto detto risulta che:

1. il controllore centrale dà inizio alle attività degli altri sottosistemi, e deve restare in ascolto di eventuali messaggi di allarme dai sottosistemi di monitoraggio;
2. i due sottosistemi di monitoraggio, dopo l'attivazione, devono funzionare continuamente e in parallelo fra di loro, interrogando periodicamente (in modo *polling*) i sensori;
3. i sensori e il sottosistema per le emergenze entrano in funzione solo quando ricevono messaggi dai sottosistemi di monitoraggio o dal controllore.

Possiamo quindi concludere che nel sistema ci sono tre flussi di controllo, appartenenti al controllore e ai due sottosistemi di monitoraggio, e iniziare il modello di progetto col diagramma di fig. 6.20, in cui abbiamo individuato gli oggetti attivi e quelli passivi.

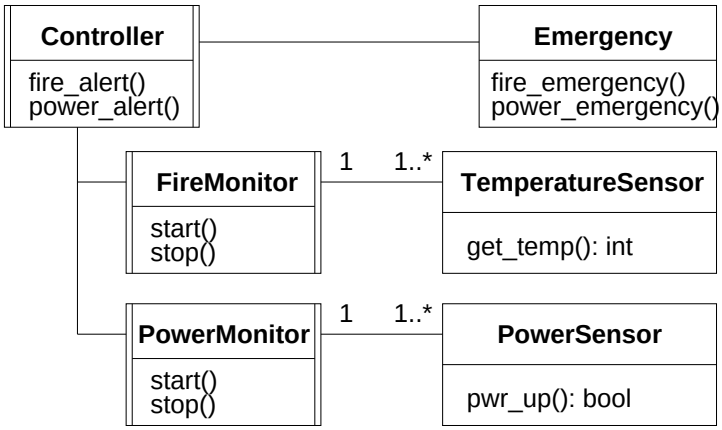


Figura 6.20: Un sistema concorrente (2).

Rappresentazione dei flussi di controllo in UML

Il linguaggio UML permette di rappresentare i flussi di controllo concorrenti in diversi modi. La fig. 6.21 mostra i flussi di controllo dell'esempio precedente per mezzo di un'estensione ai diagrammi di sequenza introdotta nell'UML2. Questa consiste nella possibilità di indicare dei segmenti di interazioni fra oggetti (cioè sottosequenze di scambi di messaggi), detti *frammenti*, che possono essere ripetuti, o eseguiti condizionalmente. In questo modo è possibile rappresentare graficamente i vari costrutti di controllo usati nei linguaggi di programmazione e anche specificare vari tipi di vincoli sulle interazioni possibili.

Nella fig. 6.21, in cui per semplicità si suppone che ci sia un solo sensore di temperatura e un solo sensore di tensione, vediamo prima di tutto un blocco etichettato par e diviso in due regioni separate da una linea tratteggiata: questo significa che le due regioni rappresentano attività che avvengono in parallelo. Infatti le due regioni si riferiscono, rispettivamente, ai due sottosistemi di monitoraggio.

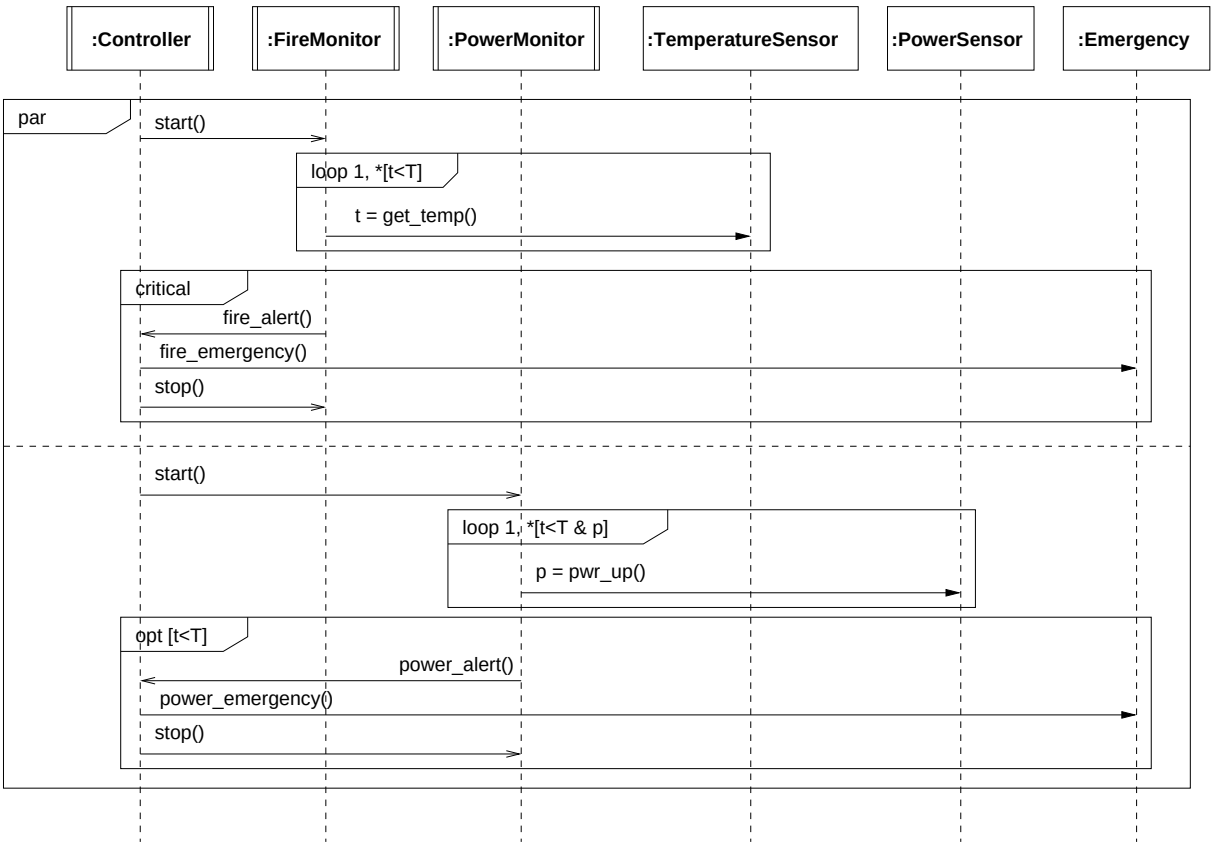


Figura 6.21: Un diagramma di sequenza con *frammenti*.

Nella prima regione, il controllore manda un messaggio start() al monitor antincendio. Questo esegue la sequenza specificata dal blocco loop, la cui espressione di controllo dice che il contenuto del blocco deve essere eseguito almeno una volta e poi essere ripetuto finché il valore t restituito da get_temp() è minore di una temperatura di soglia T. Se la temperatura di soglia viene superata, l'esecuzione del blocco loop termina, ed inizia una sequenza racchiusa in un blocco critical. Questo significa che tale sequenza (invio di un allarme al controllore, inizio della procedura di emergenza, e arresto del monitor) è una *sezione critica*, cioè un'attività non interrompibile. Questo rispecchia il requisito che il trattamento di un incendio abbia una priorità piú alta rispetto all'altra situazione di rischio.

Nella seconda regione, il controllore attiva il monitor dell'alimentazione, che esegue un blocco loop condizionato sia dal valore restituito da pwr_up() che da quello di get_temp(), poiché il ciclo deve terminare anche in caso di incendio. All'uscita dal ciclo c'è un blocco opt che viene eseguito soltanto se non è stato rilevato un incendio.

Esistono vari altri operatori che qui non tratteremo. Osserviamo infine l'uso di due tipi di frecce per due diversi tipi di messaggi: i messaggi sincroni sono destinati a oggetti passivi (i sensori e il gestore delle emergenze), quelli asincroni sono destinati a oggetti attivi.

Strumenti per la programmazione concorrente

L'implementazione degli oggetti attivi definiti nel progetto architetturale riguarda principalmente le fasi di progetto dettagliato e di codifica, ma certe scelte relative all'implementazione devono essere fatte nel progetto architetturale.

Una prima decisione riguarda la scelta fra l'uso di processi o di thread. Come già detto, un programma concorrente eseguito su un singolo calcolatore può essere composto da un solo processo multithreaded o da piú processi, ciascuno dei quali può essere multithreaded o no. Poiché la creazione e la gestione dei processi è piú costosa, in termini di tempo e risorse, di quella dei thread, generalmente si scelgono questi ultimi se non ci sono vincoli contrari.

Un'altra scelta riguarda gli strumenti da usare per realizzare i meccanismi di concorrenza richiesti. Alcuni linguaggi di programmazione, come Java e Ada, hanno dei costrutti che permettono di definire degli oggetti attivi (p.es., oggetti di classe `Thread` in Java, o moduli `task` in Ada), di controllarne l'esecuzione e di sincronizzarli. Se uno di questi linguaggi (in base a requisiti non attinenti alla concorrenza) è stato scelto per l'implementazione, ovviamente conviene sfruttare gli strumenti che offre. Se la scelta del linguaggio è ancora aperta, si possono studiare i meccanismi di concorrenza usati dal linguaggio e valutare la loro adeguatezza al caso specifico.

Altri linguaggi, come il C e il C++, non hanno istruzioni o tipi predefiniti per la concorrenza[3]. Sviluppando in questi linguaggi, si presentano queste possibilità:

- usare direttamente le chiamate di sistema;
- incapsulare le chiamate di sistema in classi sviluppate *ad hoc*;
- usare librerie di classi che implementano i meccanismi di concorrenza offrendo un'interfaccia orientata agli oggetti;
- usare librerie che implementano modelli di programmazione concorrente (o parallela) a livello piú alto dei meccanismi di base della concorrenza.

L'uso diretto delle chiamate di sistema è complesso e prono ad errori, per cui si preferiscono le altre alternative, a meno che il sistema da sviluppare non sia molto semplice, o non sia necessario avere un controllo molto fine sull'esecuzione.

Per incapsulare le chiamate di sistema bisogna scrivere delle classi le cui interfacce siano piú semplici dell'interfaccia di programmazione di sistema. Per esempio, la creazione di un thread Posix richiede l'esecuzione di un certo numero di chiamate che impostano vari parametri del thread, incluso il sottoprogramma da eseguire. Si può allora definire una classe `Thread` il cui costruttore invoca tali chiamate, per cui la creazione di un thread si riduce a istanziare questa classe. Procedendo in questo modo, si crea uno strato di

[3]Non si tratta di una dimenticanza. I creatori del C e del C++ seguono il principio che un linguaggio deve offrire un nucleo di funzioni base flessibili e potenti che si possano implementare in modo efficiente su qualsiasi piattaforma hardware e software. Le funzioni piú sofisticate, come la concorrenza o la *garbage collection*, possono quindi essere implementate in modi diversi secondo le esigenze delle diverse applicazioni. I meccanismi per la concorrenza sono disponibili nella libreria standard.

software fra l'applicazione ed il sistema operativo, che semplifica la programmazione e rende l'applicazione piú modulare e portabile.

Questa tecnica permette anche di semplificare l'accesso in *mutua esclusione* alle risorse condivise. A ciascuna di queste risorse bisogna associare un *semaforo*, cioè un meccanismo del sistema operativo che permette a un solo thread (o processo) per volta di accedervi. Un thread deve *acquisire* il semaforo, usare la risorsa, e quindi *rilasciare* il semaforo permettendo agli altri thread di accedere alla risorsa. Usando direttamente le chiamate di sistema c'è il rischio che, per un errore di programmazione, il rilascio non avvenga, con un conseguente blocco del sistema. Se il semaforo viene incapsulato in una classe, l'acquisizione viene fatta nel costruttore e il rilascio nel distruttore, cosicché il rilascio avviene automaticamente quando l'istanza del semaforo esce dal suo spazio di definizione.

Invece di implementare delle classi *ad hoc* per incapsulare i meccanismi di concorrenza, si può usare una libreria pre-esistente. Questa è probabilmente la soluzione migliore nella maggior parte dei casi. Esistono varie librerie di questo tipo, che offrono servizi simili ma con interfacce alquanto diverse e con diverse dipendenze da altre librerie e dai sistemi operativi. Citiamo, in particolare, la libreria *Boost.Thread*[4], che è stata inclusa nell'attuale versione delle librerie standard del C++ (C++11 e successive). In questa libreria, la classe `thread` ha un costruttore a cui si passa come argomento il sottoprogramma da eseguire. L'interfaccia di `thread` comprende l'operazione `join()` che permette a un thread di aspettare che termini l'esecuzione di un altro thread. Nella libreria sono definite anche funzioni globali (ma racchiuse nei rispettivi namespace) per manipolare i thread, fra cui le operazioni `sleep_for()` e `sleep_until()` che permettono al thread di sospendersi per un periodo di tempo determinato, e l'operazione `yield()` che permette al thread di cedere il proprio turno di esecuzione ad altri processi. Altre classi implementano vari meccanismi, fra cui alcuni tipi di semafori.

Infine, si può considerare l'uso di librerie ed ambienti di esecuzione che offrano un livello di astrazione piú alto. Queste librerie sono basate generalmente sul modello di programmazione a scambio di messaggi e permettono di applicare in modo abbastanza semplice numerose tecniche di programmazione parallela, in cui però non ci addentreremo. Di solito queste librerie presentano un'interfaccia costituita da una raccolta di numerose funzioni globali (in C o in Fortran), per cui anche in questo caso si può considerare l'opportunità di incapsularle in classi che offrano un'interfaccia piú strutturata. Esempi di queste librerie sono la *Parallel Virtual Machine* (*PVM*[5]) ed il *Message Passing Interface* (*MPI*[6]), destinate allo sviluppo di applicazioni parallele (cioè eseguite su processori multipli), ma applicabili anche per sistemi concorrenti non distribuiti.

Tecniche di implementazione

Il progetto di sistemi concorrenti è un argomento complesso che richiede la conoscenza di problemi e tecniche specifiche dei vari campi di applicazioni, e in questa sezione possiamo

[4] `http://www.boost.org/doc/html/thread.html`
[5] `http://www.csm.ornl.gov/pvm/`
[6] `http://www.llnl.gov/computing/tutorials/mpi/`

dare solo alcune indicazioni generiche, considerando i problemi dell'accesso alle risorse, della sincronizzazione e della comunicazione fra processi o thread.

Risorse condivise Nel progetto ad alto livello, bisogna anche stabilire, analizzando le interazioni fra sottosistemi, quali oggetti passivi vengono interessati da un solo flusso di controllo e quali sono condivisi. I primi si possono considerare come parte del corrispondente oggetto attivo, col quale possono essere messi in relazione di composizione nei diagrammi UML. I secondi devono essere protetti mediante meccanismi di mutua esclusione, fra i quali sono fondamentali i semafori di mutua esclusione (*mutex*).

Gli oggetti passivi protetti da semafori sono chiamati *monitor*. Un cliente di un monitor può accedere ai suoi dati solo attraverso le operazioni del monitor stesso, e queste provvedono ad acquisire il semaforo mettendo in attesa il chiamante se la risorsa è occupata. Queste operazioni sono dette *operazioni con guardia* o *sincronizzate*. Nel linguaggio Java, per esempio, un'operazione può essere dichiarata `synchronized`, e il compilatore le associa un semaforo e la implementa inserendo le operazioni di acquisizione e rilascio. In questo modo il programmatore non deve gestire direttamente il semaforo.

Sincronizzazione I vincoli di sincronizzazione spesso si possono esprimere in termini di eventi, stati e azioni: per esempio, si può richiedere che un oggetto attivo reagisca con una certa azione (ed eventualmente con un cambiamento del proprio stato) ad un evento causato da un altro oggetto attivo, oppure si può richiedere che un oggetto attivo resti in attesa di un evento prima di procedere nella sua esecuzione. Questo tipo di informazioni si rappresenta in modo naturale con macchine a stati associate agli oggetti attivi. A volte le macchine a stati sono già state definite nel modello di analisi, che nel modello di progetto possono essere completate o rese piú dettagliate. Un esempio di sincronizzazione fra macchine a stati che si scambiano segnali è stato visto nella fig. 4.32 (pag. 104).

Lo scambio di segnali può essere implementato per mezzo di semafori di sincronizzazione, che a loro volta, essendo risorse condivise, sono protetti da semafori di mutua esclusione.

Comunicazione La comunicazione fra oggetti attivi si può considerare come una forma particolare di sincronizzazione, in cui gli oggetti si scambiano dati. Bisogna stabilire quali comunicazioni sono *sincrone* e quali *asincrone*. In una comunicazione sincrona il mittente del messaggio si blocca finché non arriva una risposta dal destinatario, mentre in una comunicazione asincrona il mittente prosegue l'esecuzione senza aspettare la risposta, che può anche non esserci. Quando è richiesta una comunicazione sincrona, il blocco del mittente avviene automaticamente se la comunicazione avviene per mezzo di una normale chiamata di procedura, altrimenti (in un sistema a scambio di messaggi) è il meccanismo di comunicazione che deve sospendere il mittente. In ogni caso, si cerca di usare un meccanismo di comunicazione fornito dal linguaggio di programmazione, da librerie o dal sistema operativo, ma in certi casi è necessario o conveniente realizzare dei meccanismi

ad hoc. Questi sono basati su strutture dati condivise, come code o liste, che quindi richiedono le tecniche di mutua esclusione viste precedentemente.

Esempio

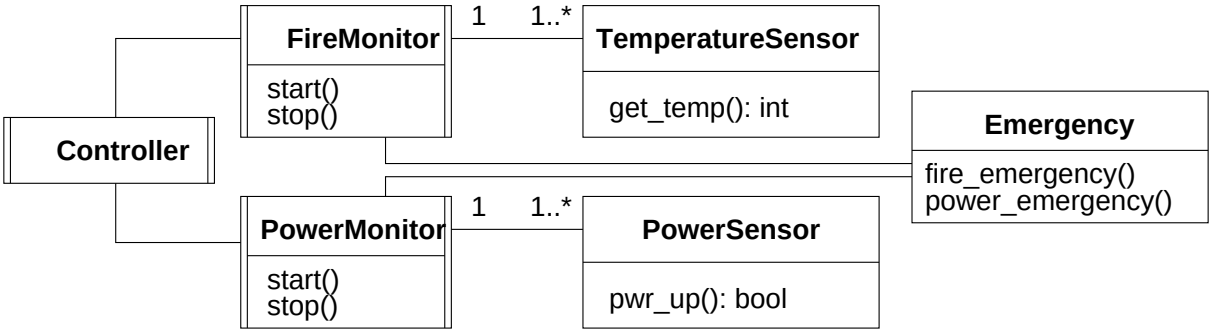


Figura 6.22: Un sistema concorrente con risorse condivise (1).

Nell'esempio di fig. 6.19 (pag. 152), ciascun oggetto passivo viene toccato dal flusso di controllo di un solo oggetto attivo: in questo modo gli oggetti attivi non condividono risorse. I rispettivi flussi di controllo non sono indipendenti perché, come abbiamo visto, l'attività di gestione di un incendio ha la priorità sulla gestione di una perdita di alimentazione, però questo vincolo viene facilmente soddisfatto programmando opportunamente il controllore, che, grazie al suo ruolo centrale, può coordinare le attività dei sottosistemi ad esso subordinati.

Se consideriamo, invece, l'architettura alternativa mostrata in fig. 6.22, vediamo che il sottosistema **Emergency** è interessato dai flussi di controllo dei due monitor (cioè, ciascuno dei due monitor può invocare operazioni di **Emergency**) ed è quindi *condiviso*. Nel progetto del sistema bisogna tener conto della condivisione di risorse per evitare problemi come il *blocco* (o *deadlock*) o l'*inconsistenza* delle informazioni causata da accessi concorrenti a dati condivisi. In questo caso particolare, bisogna rispettare il vincolo costituito dalla priorità dell'attività fire_emergency() rispetto all'altra: la gestione di una mancanza di alimentazione non può interrompere la gestione di un incendio, mentre quest'ultima deve poter interrompere la gestione della perdita di alimentazione, che a sua volta deve poter riprendere dopo che l'incendio è stato spento (nell'ipotesi che sia stato spento abbastanza presto da non fare danni).

Per ottenere questo comportamento si scompone la classe **Emergency** in due classi, separando la gestione della mancanza di alimentazione (**PowerEmergency**) da quella dell'incendio (**FireEmergency**). Questa, infatti, si comporta come un oggetto passivo, mentre l'altra, dovendo essere interrotta e riattivata in modo asincrono, è un oggetto attivo. Questa ristrutturazione dell'architettura è mostrata nella fig. 6.23, in cui la classe **channel** implementa i canali di comunicazione fra **Controller**, **FireMonitor**, **PowerMonitor** e **PowerEmergency**, implementate come thread. La classe **state** contiene le variabili di stato booleane fal_ (*fire alert*) e pal_ (*power alert*), le cui combinazioni definiscono gli

stati di funzionamento del sistema: nessun allarme, uno dei due allarmi o tutti e due gli allarmi in corso.

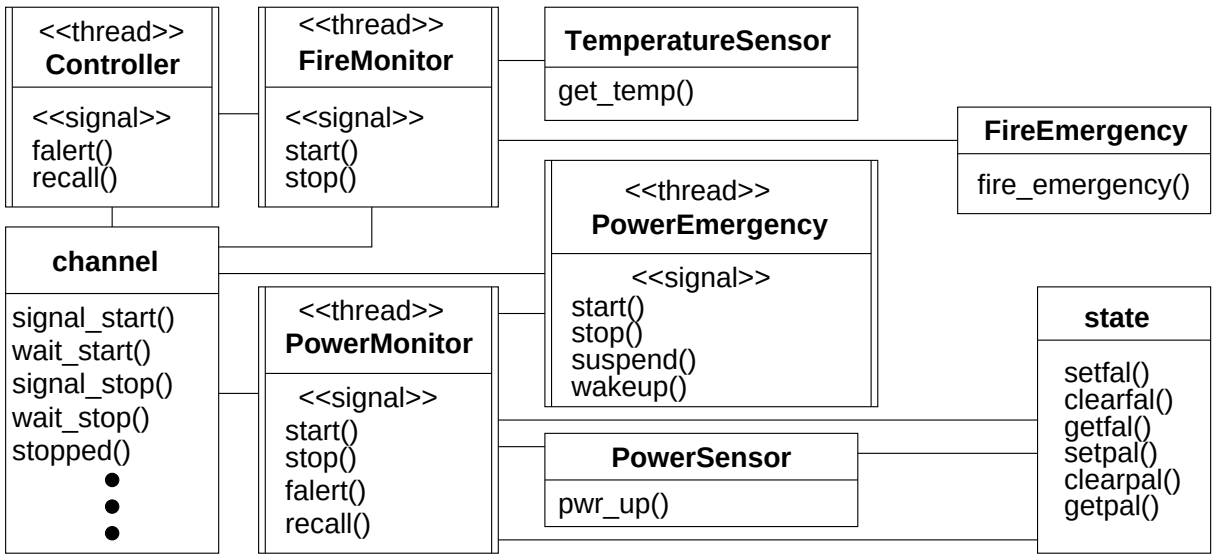


Figura 6.23: Un sistema concorrente con risorse condivise (2).

Nel diagramma di fig. 6.23, le interfacce degli oggetti attivi sono rappresentate ad alto livello in termini di segnali. I segnali start e stop vengono inviati dal controllore agli altri tre oggetti attivi. Il thread **FireMonitor** manda al controllore falert (*fire alert*) quando rileva un incendio, e recall al termine dell'emergenza. Quando il controllore riceve falert lo inoltra a **PowerMonitor** che, se **PowerEmergency** è attivo, lo mette in attesa col segnale suspend. Quando il controllore riceve recall, lo inoltra a **PowerMonitor** che, se **PowerEmergency** è sospeso, lo riattiva col segnale wakeup. Il diagramma di istanze di fig. 6.24 mostra esplicitamente i collegamenti fra i moduli, omettendo l'istanza di **state**.

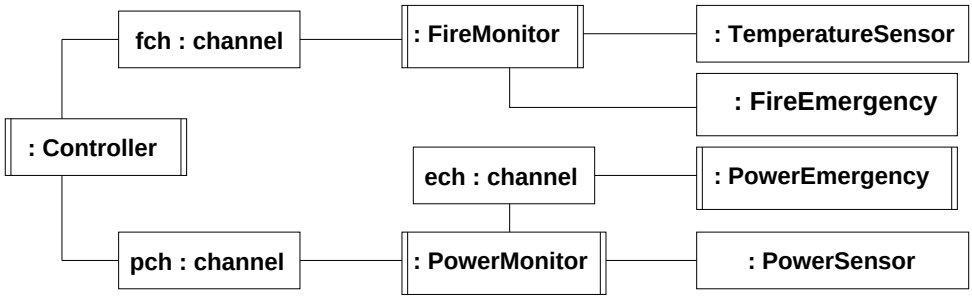


Figura 6.24: Un sistema concorrente con risorse condivise (3).

Il diagramma di sequenza di fig. 6.25 mostra uno scenario (*scenario* 1) in cui la mancanza di alimentazione viene rilevata mentre è già in corso la gestione di un incendio, per cui il ripristino dell'alimentazione viene rinviato fino al termine dell'incendio. I thread vengono attivati dal sistema operativo nel momento in cui le rispettive classi vengono istanziate, ma (eccetto **Controller**) sono programmati in modo da mettersi súbito in attesa del segnale start. L'evoluzione del sistema è descritta anche nella tab. 6.1, che ne

descrive sinteticamente gli eventi principali. sospende **PowerEmergency**, ma quest'ultimo è ancora in attesa del segnale start. Dopo averlo ricevuto (istante t_7), riconosce che è stato inviato il segnale suspend e si mette in attesa di wakeup, che arriva all'istante t_{10}.

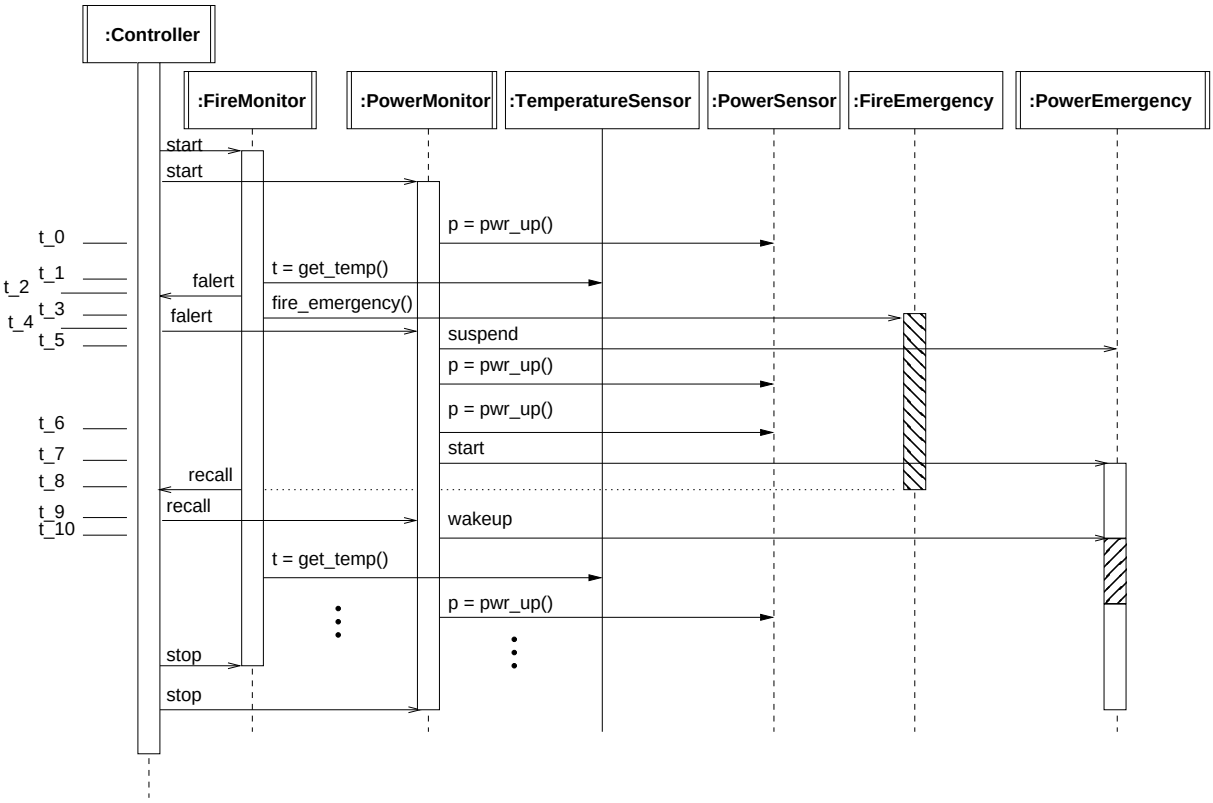


Figura 6.25: Scenario 1 per il sistema di fig. 6.23.

Un secondo scenario (*scenario* 2), in cui l'incendio viene rilevato mentre è già in corso la gestione della mancanza di alimentazione, è descritto dalla fig. 6.26 e dalla tab. 6.2. In questo caso, **PowerEmergency** viene interrotto finché non termina l'esecuzione di **FireEmergency**. Osserviamo che nel periodo fra t_4 e t_6 il thread **PowerEmergency** è attivo durante l'esecuzione di **FireEmergency**. Come utile esercizio, lo studente è invitato a modificare il sistema in modo da eliminare questo comportamento scorretto.

Per questo esempio si è scelto di implementare i segnali con meccanismi ad un livello di astrazione piuttosto basso[7], basati sui thread ed i semafori della libreria *Boost*.

Ogni segnale è rappresentato da una variabile booleana, per esempio `started_`, che viene asserita dal processo che invia il segnale. La variabile è associata ad un semaforo di sincronizzazione (`start_sem_`), chiamato *di condizione* nelle librerie Boost, e viene manipolata per mezzo di una operazione di tipo *signal* (`signal_start()`) per inviare il segnale, ed una di tipo *wait* (`wait_start()`) per mettersi in attesa del segnale:

```
class channel {
```

[7] `http://www.ing.unipi.it/~a009435/issw/nuovo_es_sist_concorrente.zip`

Tabella 6.1: Eventi nello scenario 1 per il sistema di fig. 6.23.

istante	evento
t_0	**PowerMonitor** inizia a controllare l'alimentazione.
t_1	**FireMonitor** rileva la condizione di allarme.
t_2	**FireMonitor** avverte **Controller**.
t_3	**FireMonitor** chiama `fire_emergency()`.
t_4	**Controller** avverte **PowerMonitor**.
t_5	**PowerMonitor** sospende **PowerEmergency**.
t_6	**PowerMonitor** rileva la condizione di allarme.
t_7	**PowerMonitor** fa partire **PowerEmergency**.
t_8	**FireMonitor** avverte **Controller** che `fire_emergency()` è terminata.
t_9	**Controller** avverte **PowerMonitor**.
t_{10}	**PowerMonitor** risveglia **PowerEmergency**.

Tabella 6.2: Eventi nello scenario 2 per il sistema di fig. 6.23.

istante	evento
t_0	**PowerMonitor** rileva la condizione di allarme.
t_1	**PowerMonitor** fa partire **PowerEmergency**.
t_2	**FireMonitor** rileva la condizione di allarme.
t_3	**FireMonitor** avverte **Controller**.
t_4	**FireMonitor** chiama `fire_emergency()`.
t_5	**Controller** avverte **PowerMonitor**.
t_6	**PowerMonitor** sospende **PowerEmergency**.
t_7	**FireMonitor** avverte **Controller** che `fire_emergency()` è terminata.
t_8	**Controller** avverte **PowerMonitor**.
t_9	**PowerMonitor** risveglia **PowerEmergency**.

```
    boost::mutex start_mtx_;
    boost::condition start_sem_;
    bool started_;
//  ...
public:
    channel();
    void signal_start();
    void wait_start();
//  ...
};
```

Per inviare il segnale start(), un thread esegue l'operazione `signal_start()`, che asserisce (in mutua esclusione) la variabile logica `started_` e, con `notify_one()`, segnala al semaforo di sincronizzazione `start_sem_` che può risvegliare un thread che era in attesa:

```
void
```

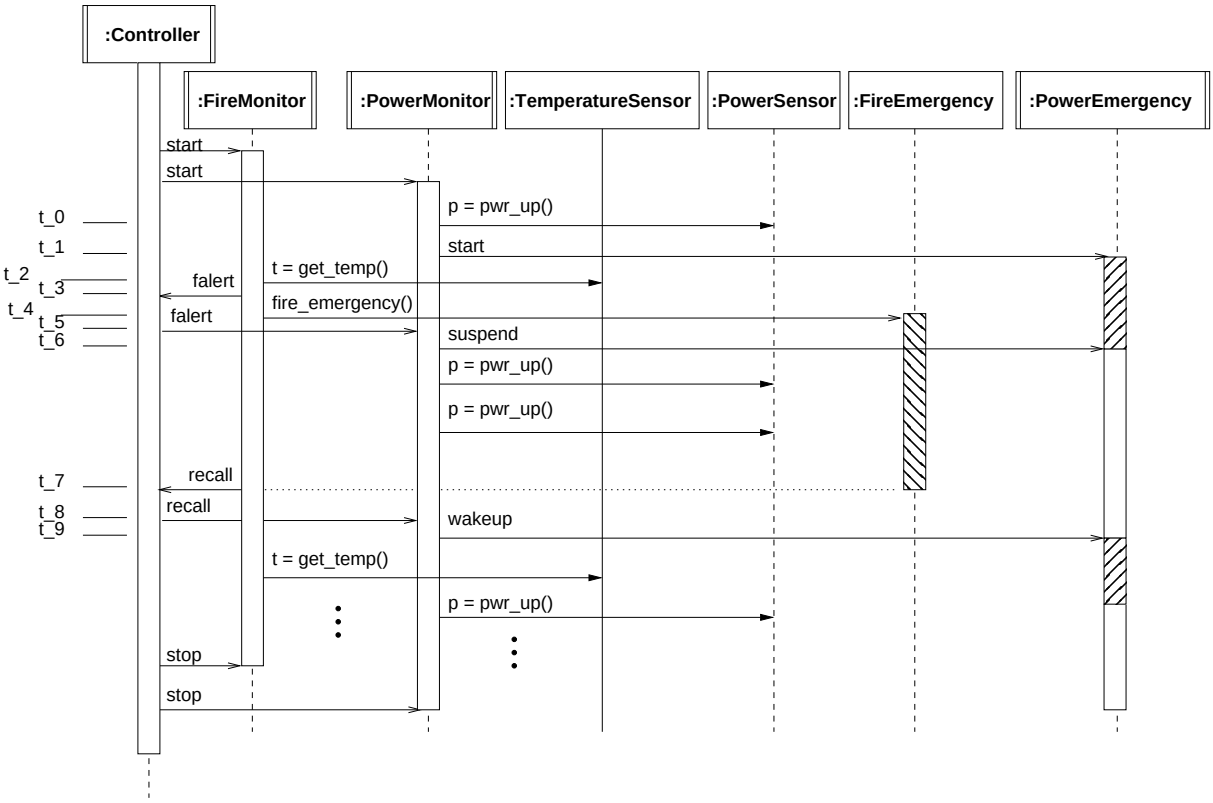


Figura 6.26: Scenario 2 per il sistema di fig. 6.23.

```
channel::
signal_start()
{
    scoped_lock lock(start_mtx_);  // acquisisce mutex
    started_ = true;               // asserisce segnale
    start_sem_.notify_one();       // sveglia un thread in attesa
}
```

La variabile `lock` è un'istanza della classe `boost::mutex::scoped_lock`[8], il cui costruttore acquisisce un semaforo di mutua esclusione che viene poi rilasciato dal distruttore.

Per aspettare il segnale start, un thread esegue l'operazione `wait_start()`, che acquisisce un *lock* sul semaforo di mutua esclusione e verifica in un ciclo `while` se il segnale è stato asserito. Se sí, esce dal ciclo e nega il segnale, altrimenti il thread si mette in attesa sul semaforo di condizione, rilasciando il lock, che verrà riacquisito quando il thread viene risvegliato:

```
void
channel::
wait_start()
{
    scoped_lock lock(start_mtx_); // acquisisce mutex
```

[8]Nel codice, il nome è stato abbreviato con un `typedef`.

```
    while (!started_)             // lettura in mutex
        start_sem_.wait(lock);    // rilascia mutex ed entra in attesa
    started_ = false;             // nega il segnale
}
```

Per alcuni segnali, per esempio `stop`, esiste anche un'operazione (`stopped`) che permette al thread ricevente di verificare se il segnale è stato inviato, senza entrare in attesa:

```
bool
channel::
stopped()
{
    scoped_lock lock(stop_mtx_);
    bool s = stopped_;
    stopped_ = false;
    return s;
}
```

Osservazione. L'esempio qui presentato ha il solo scopo di concretizzare le idee introdotte in questa sezione. Il metodo di implementazione adottato non è certo il migliore ed il progetto di sistemi concorrenti richiede ben altre conoscenze di carattere sia teorico che pratico, che vanno al di là degli obiettivi di questo corso.

6.3.4 Gestione dei dati

Nel definire la struttura globale del sistema bisogna anche scegliere come memorizzare e gestire i dati permanenti e quelli da scambiare con sistemi esterni, fra cui gli utenti. I principali aspetti da considerare sono i formati di rappresentazione ed i metodi di memorizzazione.

Formati di rappresentazione

I formati di rappresentazione si possono distinguere in *binari* e *testuali*.

Il formato binario è quello usato internamente alla memoria di lavoro ed all'unità aritmetica. I dati possono essere copiati sulla memoria permanente nello stesso formato, quando i dati memorizzati sono risultati intermedi di processi di calcolo complessi, mentre i risultati finali devono essere in un formato indipendente dall'architettura hardware su cui sono stati prodotti, e spesso devono essere leggibili da operatori umani. Per questo serve un sottosistema di *persistenza* per *serializzare* i dati convertendoli in un formato opportuno.

Nei formati testuali i dati vengono rappresentati da sequenze (o *stringhe*) di caratteri alfanumerici codificati secondo qualche standard, per esempio ASCII o Unicode. Inoltre,

i dati possono essere strutturati in base a sintassi standardizzate (come XML e derivati, o JSON) o destinate ad un'applicazione specifica. La sintassi dei dati in formato testuale viene definita per mezzo di linguaggi di specifica, come la *ASN.1*, le *grammatiche regolari* e le *grammatiche non contestuali*.

Abstract Syntax Notation La *Abstract Syntax Notation* (ASN.1) serve a descrivere i dati scambiati nei protocolli di comunicazione. I dati sono organizzati *logicamente* in vario modo (strutture, sequenze ...) a partire da tipi predefiniti (*INTEGER, BOOLEAN ...*) La specifica dei dati è *indipendente* dalla loro rappresentazione concreta. I dati specificati in ASN.1 possono essere rappresentati concretamente in vari modi, descritti da *regole di codifica* (*encoding rules*). Il seguente esempio è la specifica in ASN.1 dei pacchetti di dati trasmessi da un'ipotetica stazione meteorologica:

```
WeatherReport ::= SEQUENCE
{
  stationNumber  INTEGER (1..99999),  -- sottoinsieme
  timeOfReport   UTCTime,             -- ora e data formato UTC
  pressure       INTEGER (850..1100),
  temperature    INTEGER (-100..60),
}
```

Grammatiche regolari Le grammatiche regolari specificano la forma degli elementi lessicali (*token*), cioè le parole, di un linguaggio, per mezzo di *espressioni regolari*. Dato un alfabeto finito di simboli, un'espressione regolare definisce un insieme di sequenze di simboli. Un'espressione regolare è formata da simboli e operatori, fra cui la *concatenazione*, la *ripetizione* (piú correttamente, la *chiusura*) e la *scelta*. Programmi come il *Lex*[9] producono il codice che riconosce se una stringa di caratteri appartiene all'insieme definito da un'espressione regolare.

Il seguente frammento in linguaggio Lex specifica i token che possono apparire in un'espressione aritmetica. Le parentesi quadre rappresentano insiemi di caratteri, l'asterisco (*chiusura di Kleene*) rappresenta zero o piú occorrenze consecutive della sottoespressione precedente, mentre il segno `'+'` ne rappresenta una o piú. Il frammento è formato da sette regole, composte da un'espressione regolare ed un'azione da eseguire quando viene riconosciuta una stringa appartenente all'insieme corrispondente. Nella prima regola, viene riconosciuta come identificatore ogni stringa formata da un carattere minuscolo (`a-z`) o maiuscolo (`A-Z`) o un trattino basso (`_`), seguito da zero o piú caratteri minuscoli o maiuscoli, o trattini bassi, o cifre decimali (`0-9`).

```
[a-zA-Z_][a-zA-Z_0-9]*    return( IDENTIFIER );
[0-9]+                    return( INTEGER );
"="                       return( '=' );
```

[9] `http://www.cs.utexas.edu/users/novak/lexpaper.htm`

```
"+"                     return( '+' );
"-"                     return( '-' );
"*"                     return( '*' );
"/"                     return( '/' );
```

Il programma Lex trasforma queste regole in una funzione in linguaggio C che legge un testo e per ogni token riconosciuto restituisce un valore che identifica il tipo di token, come indicato dall'azione di ciascuna regola. Le azioni possono eseguire anche altre operazioni, come gestire una tabella dei simboli.

Grammatiche non contestuali Le grammatiche non contestuali (*context-free*) specificano la sintassi di un linguaggio, cioè i modi possibili di combinarne i token. Dato un alfabeto finito di *simboli terminali* (token), una *produzione* è una regola (anche *ricorsiva*) che definisce un *simbolo non terminale*, cioè un insieme di sequenze di simboli terminali e non terminali. Per esempio, nei linguaggi di programmazione i simboli terminali sono gli *identificatori*, le *parole chiave* etc. I simboli non terminali sono i vari tipi di istruzioni. Programmi come lo *Yacc*[10] producono il codice che riconosce se una sequenza di simboli appartiene all'insieme definito da una produzione. Le grammatiche non contestuali servono anche a definire linguaggi per la rappresentazione di dati *strutturati*, come l'XML o il JSON.

Il seguente frammento in linguaggio Yacc specifica le possibili strutture di un'espressione aritmetica, definite da sei produzioni alternative. Sono state omesse le azioni associate alle produzioni, che generalmente servono a costruire una struttura dati ad albero che rappresenta il testo.

```
expr : expr '+' expr
     | expr '-' expr
     | expr '*' expr
     | expr '/' expr
     | IDENTIFIER
     | INTEGER
     ;
```

Metodi di memorizzazione

Una delle scelte principali è fra l'uso del filesystem offerto dal sistema operativo e l'uso di un sistema di gestione di database. Quest'ultima soluzione è generalmente preferibile quando la gestione di dati costituisca una parte importante, per dimensioni e complessità, delle funzioni offerte dal sistema. La scelta di usare un gestore di database comporta scelte ulteriori, come quella fra sistemi relazionali e sistemi orientati agli oggetti.

[10] `http://www.cs.utexas.edu/users/novak/yaccpaper.htm`

Oltre ai database relazionali e quelli orientati agli oggetti, esistono numerosi sistemi per la gestione dei dati. Citiamo, per esempio, i sistemi basati sul protocollo LDAP (Lightweight Directory Access Protocol)[11], usati per i servizi di directory (database gerarchici), ed i sistemi basati sul formato HDF5 (Hierarchical Data Format)[12], usati per applicazioni scientifiche.

Un altro aspetto da considerare è l'opportunità di *replicare* i dati per ragioni di tolleranza ai guasti e di accessibilità. Se i dati sono replicati, bisogna affrontare i problemi relativi alla consistenza dei dati fra le varie copie, e dell'accesso concorrente alle stesse.

Ovviamente, tutto questo è legato strettamente al supporto fisico dei dati, che comprende uno spettro molto ampio di scelte, determinate dalla quantità di dati, dai tempi di accesso, dai requisiti di affidabilità, dai tempi previsti per la loro conservazione, e via dicendo. Esistono sistemi dedicati alla memorizzazione di grandi quantità di dati che dispongono del loro specifico software di gestione da integrare nel sistema complessivo.

6.4 Architettura fisica

Per *architettura fisica* si intende l'insieme dell'*architettura fisica del software* e dell'*architettura hardware*. L'architettura fisica del software è costituita dai *componenti software fisici*, o *artefatti*, che sono il prodotto finale del processo di sviluppo, cioè i file necessari per il funzionamento del sistema sviluppato. L'architettura hardware è costituita dai *nodi* che eseguono i componenti software.

Gli artefatti possono essere file eseguibili, librerie, file sorgente, file di configurazione, pagine web, o altre cose ancora. Vengono rappresentati in UML come rettangoli con lo stereotipo ≪artifact≫. Poiché gli artefatti contengono l'implementazione di elementi logici, come classi e componenti, è utile esprimere la relazione fra questi elementi logici e gli artefatti: si dice che un artefatto *manifesta* certi elementi logici, e questo si rappresenta graficamente con una dipendenza diretta dall'artefatto agli elementi manifestati, etichettata con lo stereotipo ≪manifest≫ (fig. 6.27).

I nodi modellano, generalmente, dei singoli calcolatori che possono essere collegati in rete. Se fosse necessario specificare dettagliatamente la struttura del calcolatore, allora un nodo può rappresentare parti di un calcolatore, come la CPU o le unità periferiche, ma questo non è comune.

I *diagrammi di deployment* descrivono l'architettura fisica mostrando i nodi, i loro collegamenti, e gli artefatti istallati sui nodi. L'istallazione di un artefatto su un nodo può essere raffigurata disegnando l'artefatto dentro al nodo, o usando la dipendenza ≪deploy≫ diretta dall'artefatto al nodo (fig. 6.28).

Un nodo considerato solo come hardware viene chiamato *dispositivo* e viene rappresentato come in fig. 6.28, etichettato con lo stereotipo ≪device≫. Oltre a modellare la

[11] `http://ldap.com`
[12] `http://www.hdfgroup.org`

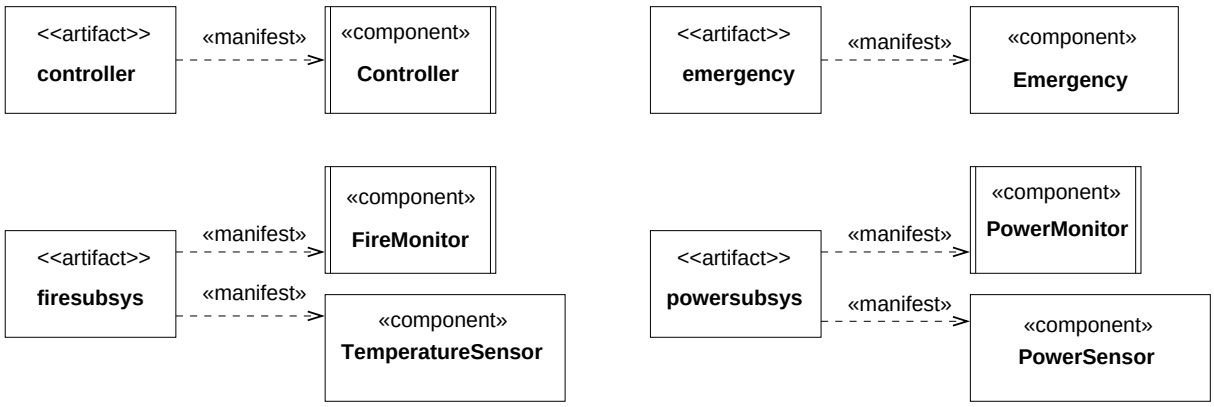


Figura 6.27: Artefatti e componenti.

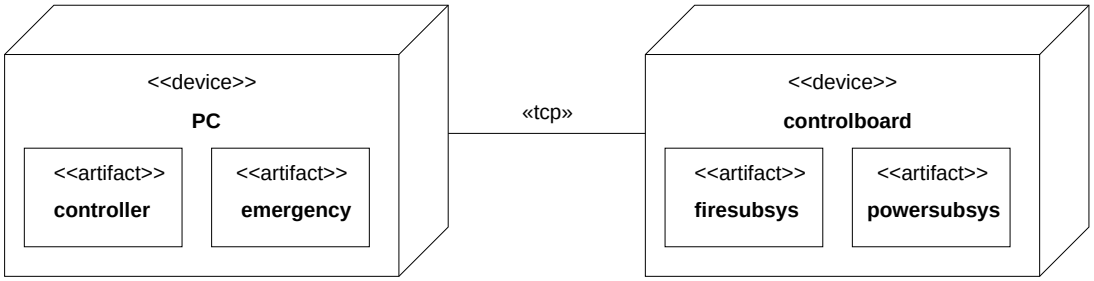


Figura 6.28: Architettura fisica.

parte hardware di un sistema, un nodo può rappresentare anche un *ambiente di esecuzione*, cioè un sistema software, esterno all'applicazione sviluppata, entro cui viene eseguita l'applicazione. Un esempio ovvio di ambiente di esecuzione è il sistema operativo, che però, generalmente, non viene modellato esplicitamente. Un sistema che generalmente viene modellato esplicitamente come ambiente di esecuzione è un server web, visto come piattaforma su cui vengono eseguiti programmi *plug-in*, cioè sviluppati separatamente ed eseguiti dal server su richiesta. Gli ambienti di esecuzione si rappresentano come nodi con lo stereotipo ≪execution environment≫ (fig. 6.29).

6.5 I *Design pattern*

Nel progetto dei sistemi software esistono dei problemi che si presentano entro svariati campi di applicazione con diverse forme e varianti, ma con una struttura comune. Per questi problemi esistono delle soluzioni tipiche la cui utilità è stata verificata con l'esperienza di numerosi sviluppatori nell'àmbito di progetti diversi. Tali soluzioni sono chiamate *design pattern* [15], ed esiste una letteratura abbastanza vasta che elenca numerosi pattern, fornendo allo sviluppatore un ricco armamentario di strumenti di progetto.

Per esempio, la fig. 6.30 mostra la soluzione di due problemi simili: modellare espressioni aritmetiche che possono contenere altre espressioni aritmetiche, e modellare elementi

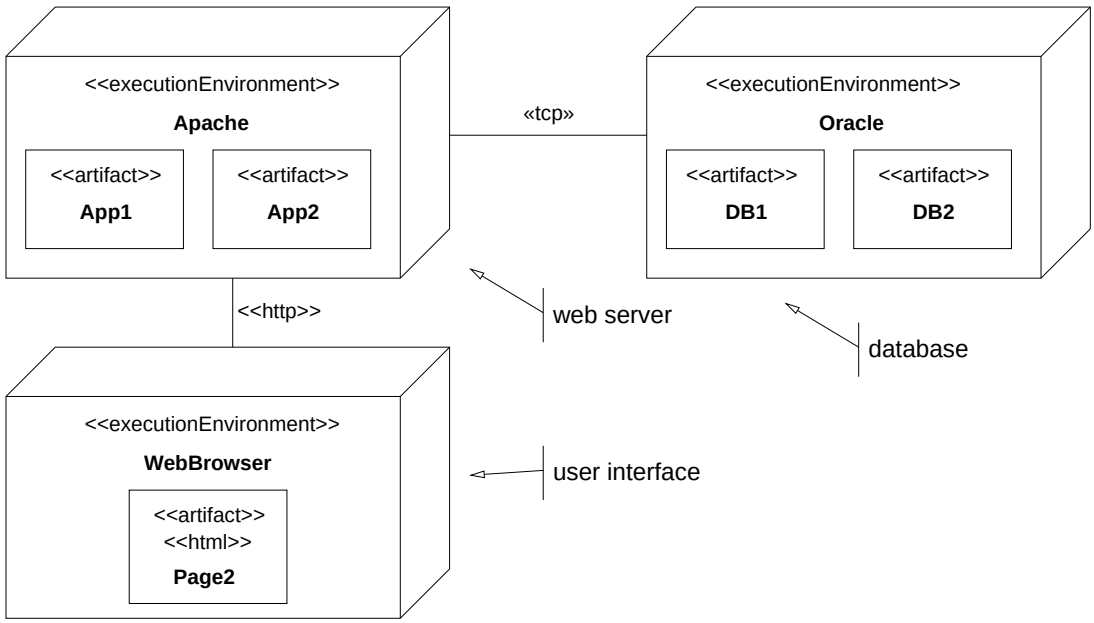


Figura 6.29: Ambienti di esecuzione.

grafici di un'interfaccia utente che possono contenerne altri. La figura mette in evidenza la similitudine fra le due strutture, e gli elementi comuni di queste due strutture si possono rappresentare astrattamente come in fig. 6.31, cioè col pattern *Composite*.

Un pattern consiste nella descrizione sintetica di un problema e della relativa soluzione. Questa viene descritta in forma sia grafica che testuale elencandone gli elementi strutturali (classi, componenti, interfacce...) con le relazioni reciproche e il loro modo di interagire. Questa descrizione viene integrata con una discussione dei vantaggi e svantaggi della soluzione, delle condizioni di applicabilità e delle possibili tecniche di implementazione. Normalmente vengono presentati anche dei casi di studio.

Un design pattern *non* è un componente software, ma solo uno schema di soluzione

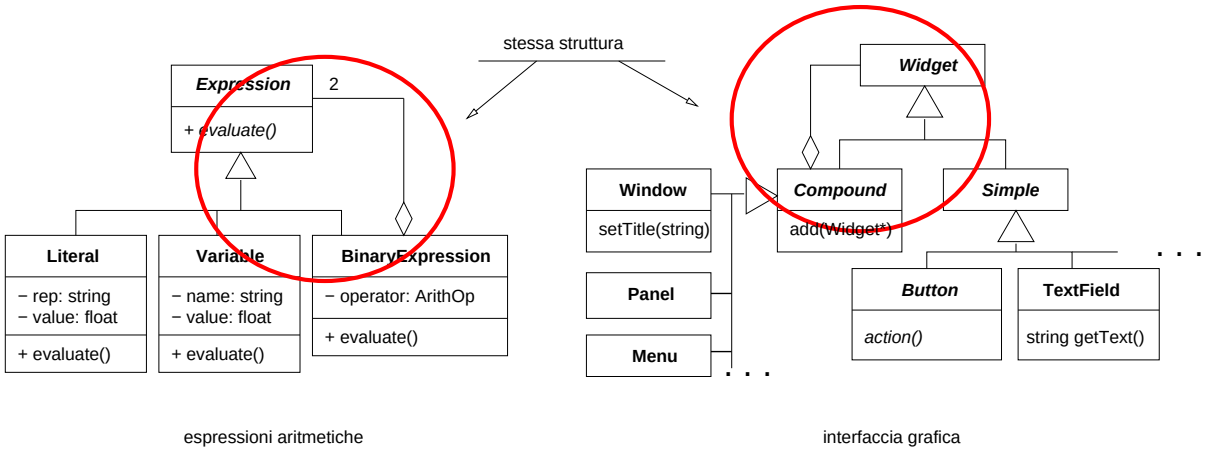


Figura 6.30: Due problemi simili.

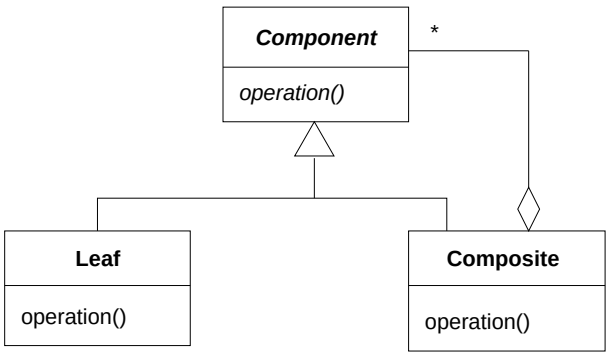


Figura 6.31: Design pattern *Composite*.

per un particolare aspetto del funzionamento di un sistema. Gli elementi strutturali di un pattern rappresentano dei *ruoli* che saranno interpretati dalle entità effettivamente realizzate. Ciascuna di queste entità può interpretare un ruolo diverso in diversi pattern, poiché in un singolo sistema si devono risolvere diversi problemi con diversi pattern. In fig. 6.32, la classe **BinaryExpression** gioca il ruolo di **Composite** nel pattern Composite e ricorre al pattern Adapter (*infra*) per implementare l'operazione evaluate, assumendo in questo pattern il ruolo di **Client**. Inoltre, è bene tener presente che un pattern non è una ricetta rigida da applicare meccanicamente, ma uno schema che deve essere adattato alle diverse situazioni, anche con un po' d'inventiva.

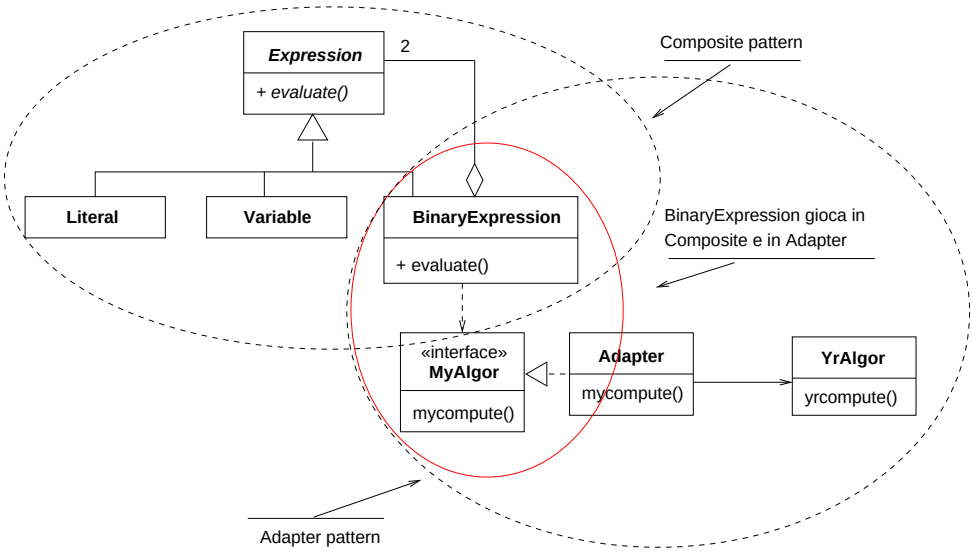


Figura 6.32: Sovrapposizione di design pattern.

I design pattern tendono ad essere usati in una fase del progetto, detta *progetto dei meccanismi* (*mechanistic design*), intermedia fra il progetto di sistema e quello in dettaglio, però molti pattern sono applicabili in tutto l'arco della fase di progetto, e inoltre esistono dei pattern concepiti espressamente per il progetto di sistema, fra cui le architetture citate nella sez. 6.3.1.

6.5.1 Composite

Come anticipato nella sezione precedente, il pattern Composite risolve il problema di organizzare un sistema formato da elementi diversi ma derivati da un'interfaccia comune, che possono contenere altri elementi derivati dalla stessa interfaccia. La soluzione consiste nel definire l'interfaccia comune in una classe astratta o un elemento ≪interface≫ chiamata *Component* nel pattern (fig. 6.31), da cui derivano classi cui appartengono elementi semplici rappresentate dalla classe **Leaf** e le classi cui appartengono elementi composti, rappresentate dalla classe **Composite**. Questa è un'aggregazione di oggetti *Component*. L'operazione *operation* rappresenta astrattamente l'insieme delle operazioni appartenenti a *Component*.

6.5.2 Adapter

Questo pattern ha lo scopo di adattare un'interfaccia offerta ad un'interfaccia richiesta.

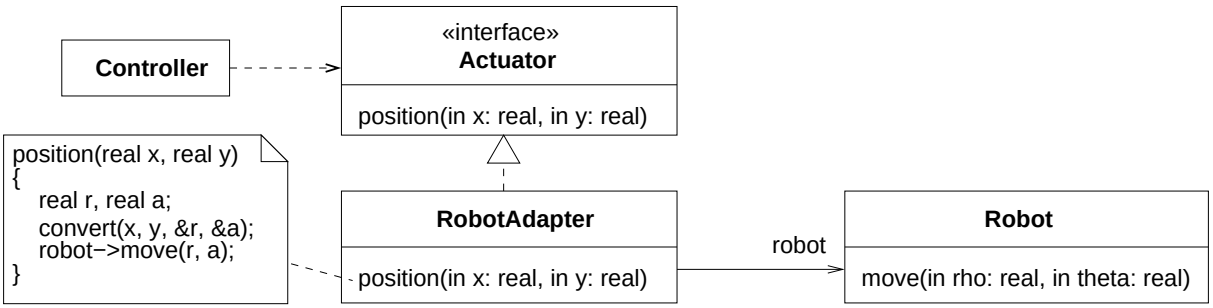


Figura 6.33: Problema: adattare l'interfaccia offerta a quella richiesta.

In fig. 6.33, la classe **Controller** richiede l'interfaccia **Actuator**. La semantica dell'operazione astratta position() è "posizionare un utensile sul punto di coordinate cartesiane (x, y)". L'interfaccia offerta dalla classe **Robot** ha l'operazione move() che posiziona l'utensile sul punto di coordinate polari (ρ, θ). Le due classi si possono adattare inserendo la classe **RobotAdapter** che realizza **Actuator** trasformando le coordinate cartesiane in polari, e chiamando l'operazione move().

Il pattern *Adapter*, detto anche *Wrapper*, generalizza questa soluzione, come in fig. 6.34.

6.5.3 Proxy

Il pattern *Proxy* viene applicato in diversi tipi di problemi, fra cui quello di risparmiare tempo di esecuzione controllando gli accessi a grandi strutture di dati (*Proxy virtuale*), e quello di interagire con un oggetto remoto come se fosse locale (*Proxy remoto*).

Proxy virtuale Lo scopo di questo pattern è differire operazioni costose.

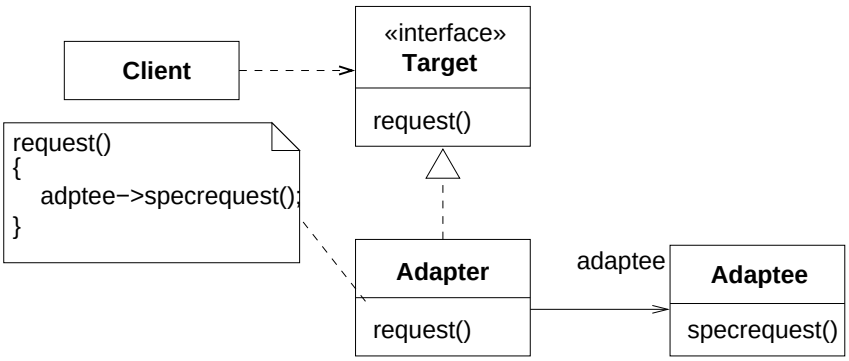


Figura 6.34: Design pattern *Adapter*.

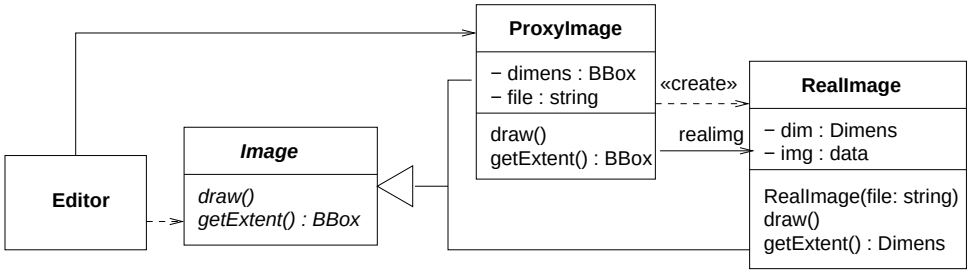


Figura 6.35: Problema: differire operazioni costose.

Per esempio, un programma di videoscrittura deve inserire nel testo delle immagini, implementate da istanze di una classe **Image** che offre le operazioni draw(), che carica in memoria l'immagine e la disegna, e getExtent(), che restituisce le dimensioni dell'immagine. Per impaginare il testo basta che siano note le dimensioni delle immagini, quindi conviene differire il caricamento dell'immagine fintanto che non è necessario visualizzarla.

Questo si ottiene (fig. 6.35) usando un'istanza della classe **ProxyImage** facente da segnaposto per **RealImage**, che contiene una struttura dati (img) per rappresentare l'immagine. Le chiamate all'operazione getExtent() vengono eseguite direttamente da **ProxyImage**, mentre le chiamate a draw() vengono delegate a **RealImage**, che viene istanziata solo alla prima invocazione di draw().

Il pattern corrispondente è mostrato in fig. 6.36.

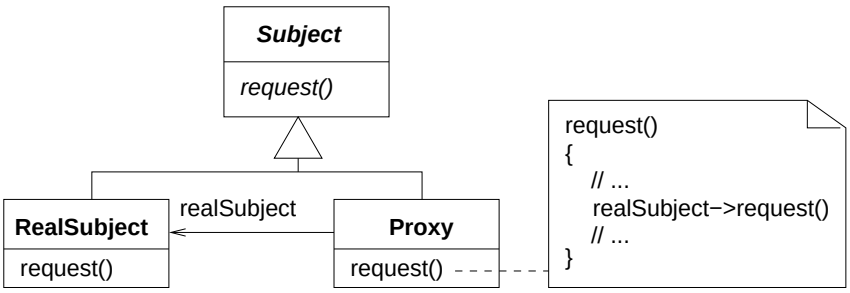


Figura 6.36: Design pattern *Proxy virtuale*.

Proxy remoto Il proxy remoto, fondamentale nelle applicazioni distribuite, permette di chiamare da remoto le operazioni di un oggetto.

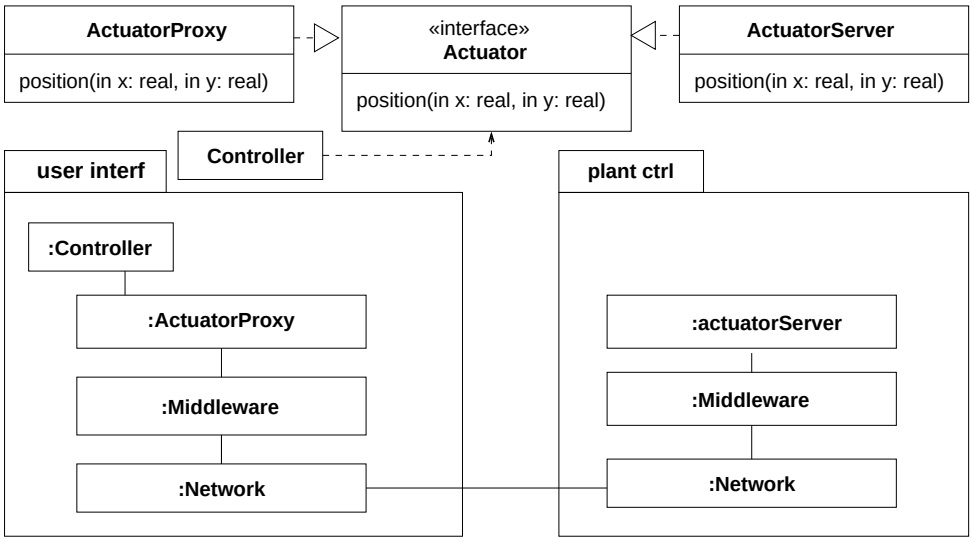


Figura 6.37: Problema: accedere a oggetti remoti.

Consideriamo, per esempio, un impianto che deve essere comandato remotamente. L'interfaccia utente (locale) contiene un'istanza della classe **ActuatorProxy** che realizza l'interfaccia di programmazione dell'impianto da controllare. Le operazioni di **ActuatorProxy** inviano messaggi al sistema di controllo remoto, usando un'infrastruttura di comunicazione. Dal lato dell'impianto, i messaggi vengono convertiti in chiamate a un'istanza di **ActuatorServer**. (V. anche `http://www.ing.unipi.it/~a009435/issw/extra/corba0708.pdf`).

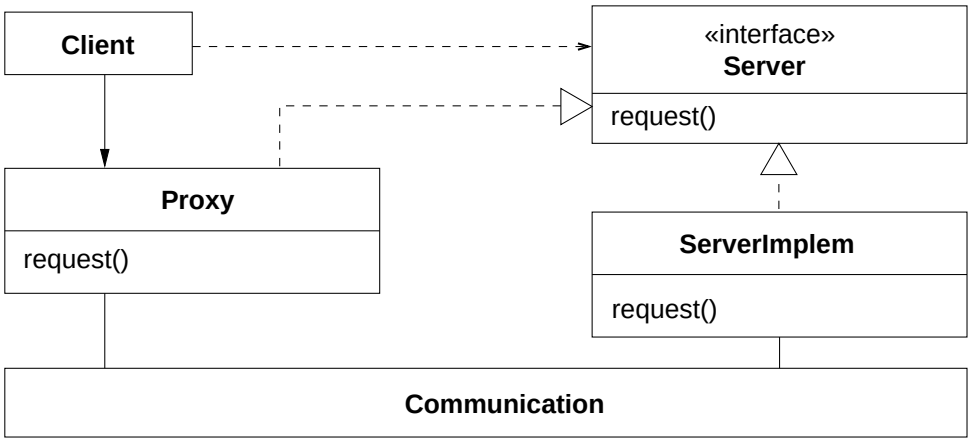


Figura 6.38: Design pattern *Proxy remoto*.

La fig. 6.37 mostra il diagramma di classi e quello di collaborazione. L'infrastruttura di comunicazione è scomposta in due strati, **Middleware** orientato allo scambio di messaggi (p.es., lo *Object Request Broker* dello standard CORBA), e **Network** rappresentante il sistema di comunicazione a livello di rete (p.es., TCP/IP).

La fig. 6.38 mostra la struttura del pattern.

6.5.4 Bridge

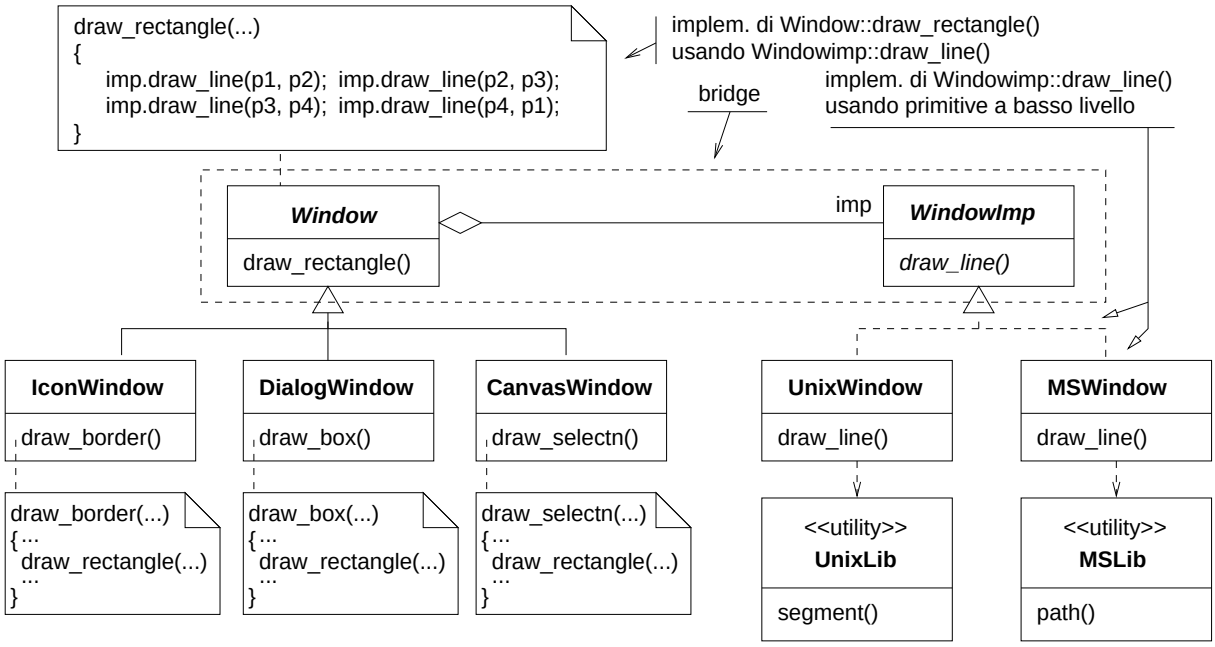


Figura 6.39: Problema: disaccoppiare una famiglia di classi dalle implementazioni.

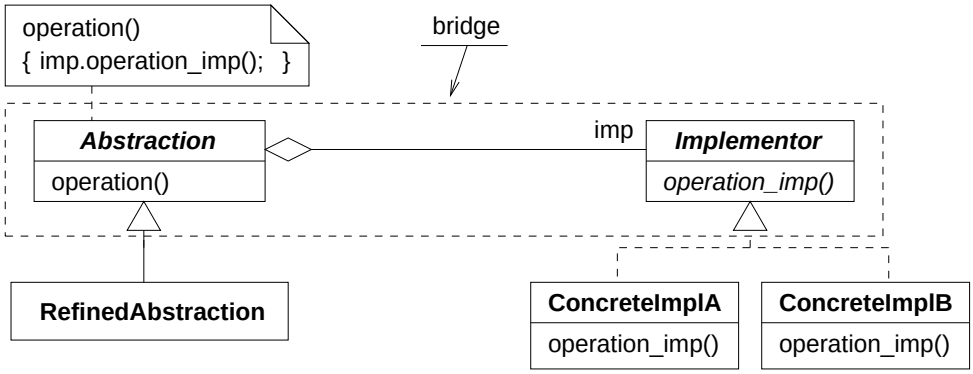


Figura 6.40: Design pattern *Bridge*.

Il pattern Bridge serve a disaccoppiare una famiglia di classi dalla loro implementazione, in modo da svilupparla indipendentemente.

Per esempio, un'interfaccia grafica comprende una famiglia di classi derivate dalla classe (astratta o concreta) *Window*; vogliamo poter implementare l'interfaccia grafica usando diverse librerie grafiche, e poterla modificare indipendentemente dalla libreria adottata. Nella soluzione proposta (fig. 6.39), la classe *Window*, che offre operazioni ad alto livello (p.es., draw_rectangle()) dipende da una classe astratta (o interfaccia) *WindowImp*

con operazioni a piú basso livello (p.es., draw_line()). Le operazioni di *Window* e delle classi derivate sono implementate con operazioni (astratte) di *WindowImp*. Queste sono implementate con le operazioni delle classi **UnixWindow** e **MSWindow**, che a loro volta dipendono dalle librerie grafiche **UnixLib** e **MSLib**.

Nel pattern Bridge, mostrato in fig. 6.40, le classi *Abstraction* e **RefinedAbstraction** rappresentano la famiglia di classi (**Window** e derivate nell'esempio) che si vuole disaccoppiare dall'implementazione, mentre *Implementor* rappresenta l'implementazione astratta (*WindowImp*) con le implementazioni concrete alternative (**UnixWindow** ed **MSWindow**).

6.5.5 Abstract Factory

Questo pattern permette di progettare una famiglia di classi astratte interrelate, in modo che tale famiglia si possa implementare e istanziare scegliendo fra diverse librerie o framework.

Supponiamo (fig. 6.41) che un'applicazione acceda a un database usando le interfacce **Connection** e **Credentials**. Queste possono essere implementate da due database diversi, XDB e YDB, scelte dall'applicazione all'inizio dell'esecuzione. Inoltre, si chiede che l'applicazione non debba, prima di ogni operazione di accesso, controllare il tipo di database che è stato scelto.

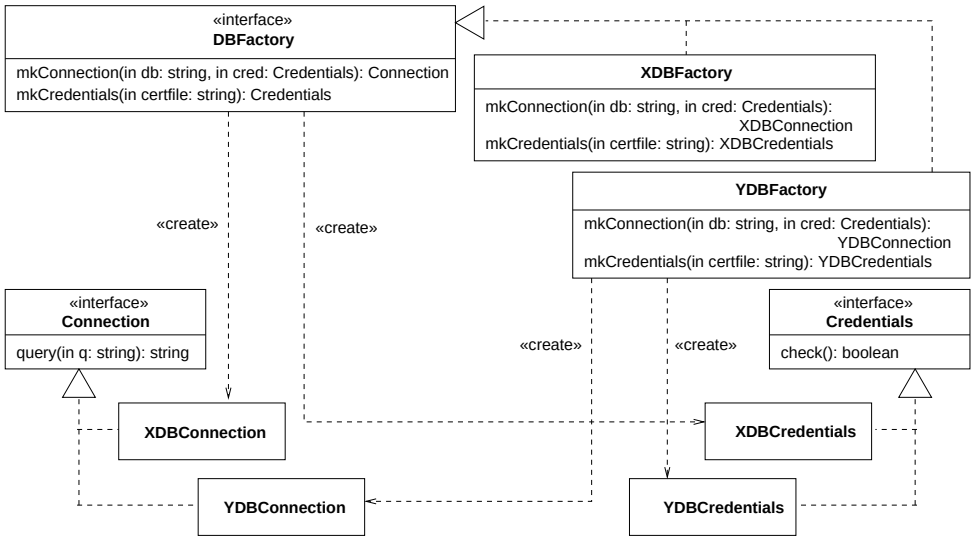


Figura 6.41: Problema: istanziare implementazioni alternative di una famiglia di classi.

La fig. 6.41 mostra una soluzione di questo problema, in cui si definisce un'interfaccia **DBFactory** per creare istanze di **Connection** e **Credentials**, realizzata dalle classi **XDBFactory** e **YDBFactory**, una per ciascuna delle librerie alternative.

Il pattern corrispondente è mostrato in fig. 6.42.

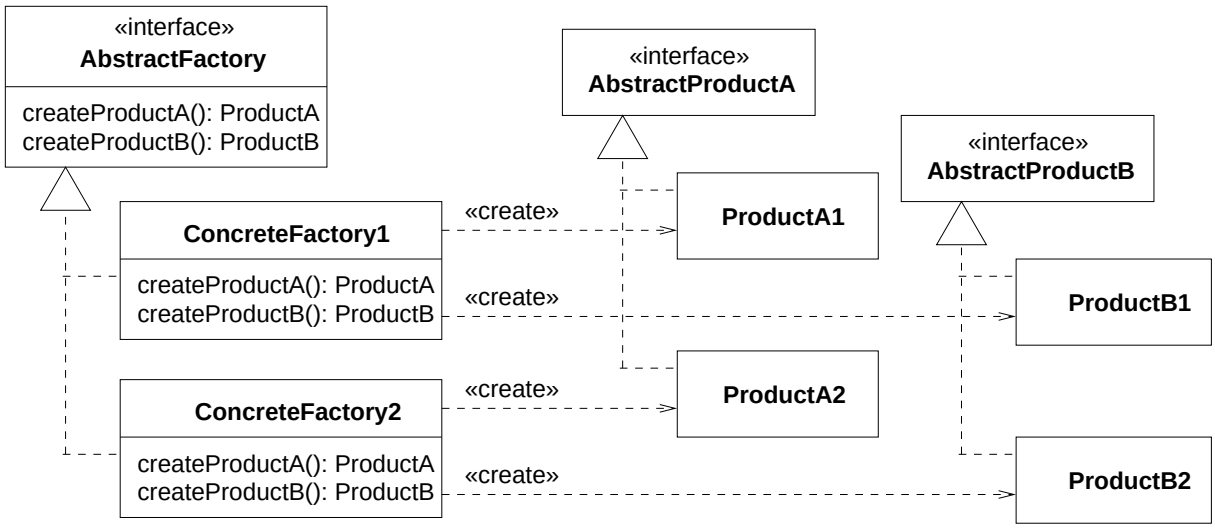


Figura 6.42: Design pattern *Abstract Factory*.

Osserviamo che il pattern Bridge visto precedentemente risponde al problema di *progettare* una famiglia di classi indipendentemente dalle possibili implementazioni, mentre il pattern Abstract Factory serve a *istanziare* implementazioni alternative di una famiglia di classi.

6.5.6 Iterator

Il pattern Iterator permette di accedere in sequenza agli elementi di una struttura dati senza dipendere dalla sua implementazione.

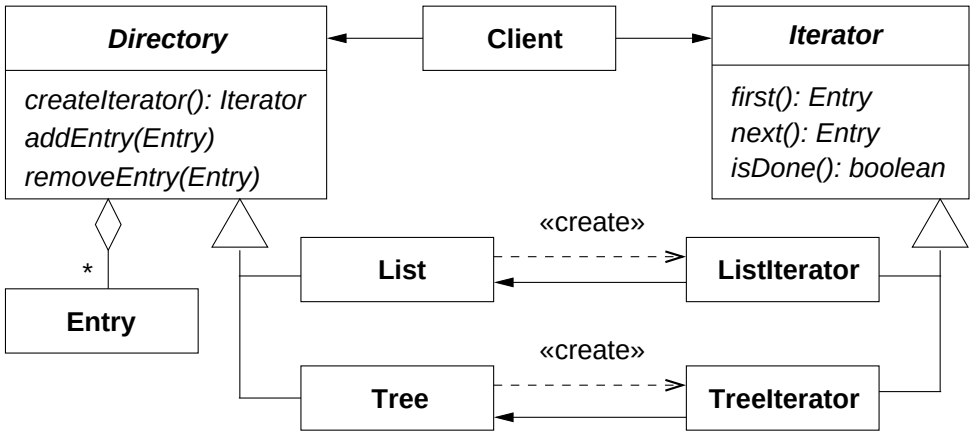


Figura 6.43: Problema: accesso uniforme a contenitori diversi.

Per esempio, un elenco di voci può essere implementato usando strutture dati diverse, come liste (**List**) o alberi (**Tree**). Un'applicazione deve poter accedere agli elementi di questo elenco indipendentemente dalla struttura in cui viene implementato. A questo scopo (fig. 6.43) si definisce una classe astratta (o un'interfaccia) *Iterator* che offre le

operazioni necessarie per accedere in sequenza alle voci (**Entry**) dell'elenco (**Directory**). Queste operazioni vengono implementate da classi specifiche per ciascuna struttura dati (**ListIterator** e **TreeIterator**). La classe astratta (o interfaccia) **Directory** offre le operazioni per costruire l'elenco e per creare un iteratore, e viene implementata da **List** o **Tree**.

Il pattern corrispondente è mostrato in fig. 6.44, dove la classe *Aggregate* rappresenta il ruolo di *Directory* nell'esempio, mentre **ConcreteAggregate1** e **ConcreteAggregate2** rappresentano il ruolo delle sue implementazioni alternative.

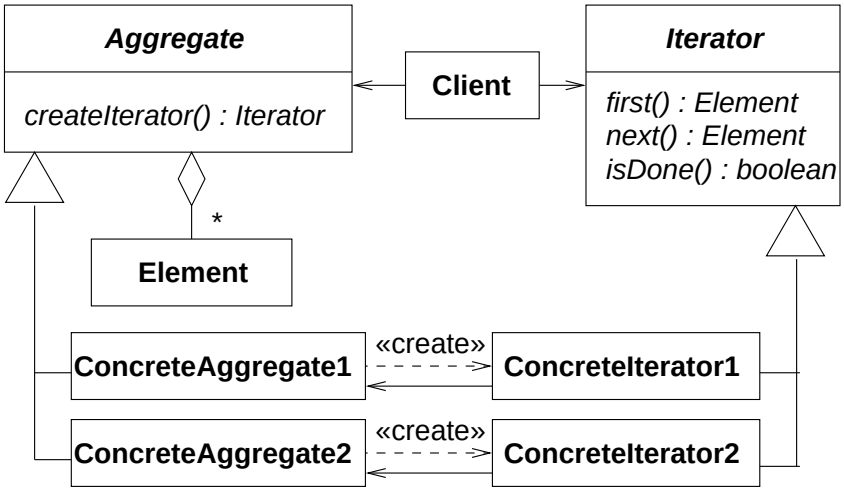


Figura 6.44: Design pattern *Iterator*.

6.5.7 Strategy

Il pattern Strategy serve a definire una famiglia di algoritmi intercambiabili.

Per esempio, consideriamo un programma di videoscrittura che deve (re)impaginare un testo dopo le modifiche fatte dall'utente, usando algoritmi di impaginazione alternativi. Ricordiamo che un algoritmo di impaginazione deve riempire la pagina in modo uniforme, tenendo conto delle spaziature, delle dimensioni dei caratteri, delle illustrazioni eccetera.

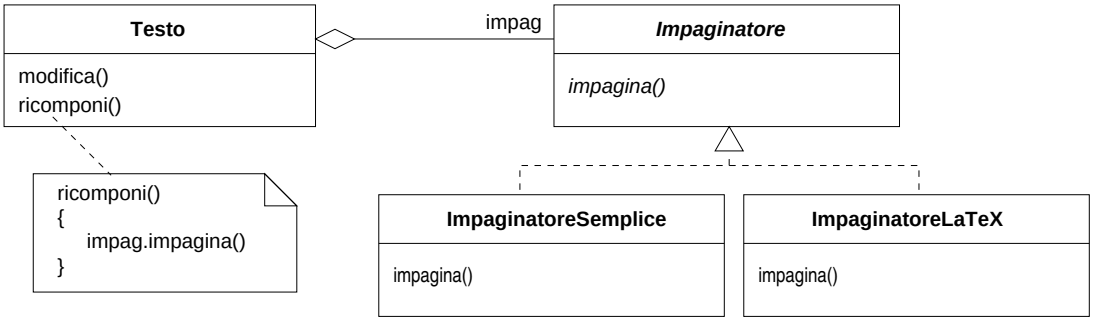


Figura 6.45: Problema: disporre di algoritmi intercambiabili.

Per risolvere questo problema (fig. 6.45), si definisce un'interfaccia comune (*Impaginatore*) per i diversi algoritmi, implementata da classi diverse (**ImpaginatoreSemplice** ed **ImpaginatoreLaTex**). La classe **Testo** offre le operazioni per modificare il testo e per reimpaginarlo.

La fig. 6.46 mostra la struttura del pattern che astrae questa soluzione.

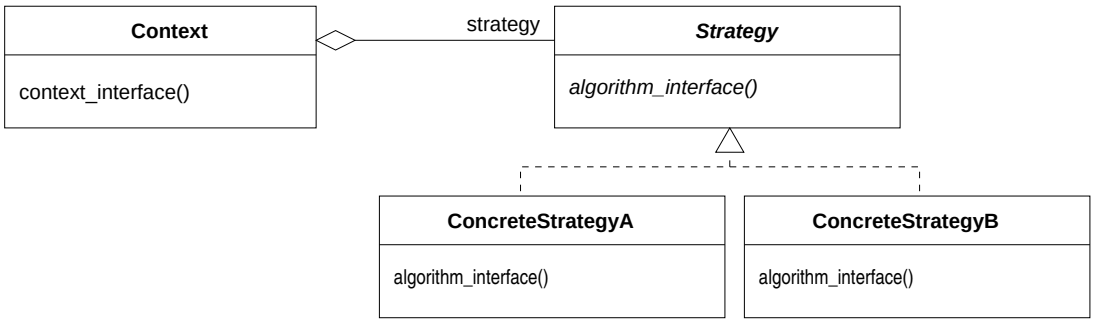


Figura 6.46: Design pattern *Strategy*.

6.5.8 Observer

Questo pattern serve a fare in modo che un oggetto possa notificare altri oggetti dei suoi cambiamenti di stato.

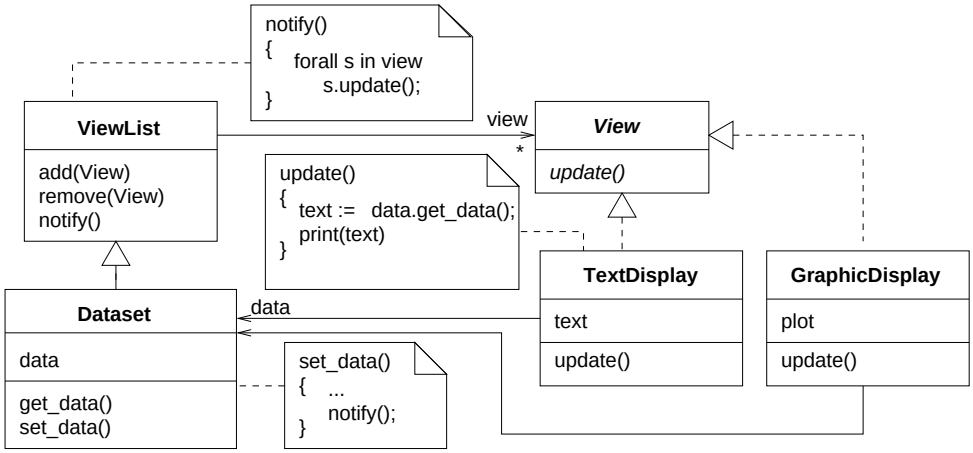


Figura 6.47: Problema: notificare ai clienti i cambiamenti di stato.

Nell'esempio di fig. 6.47, un insieme di dati (istanza di **Dataset**) deve poter essere visualizzato in forma testuale ed in forma grafica.

Si definisce un'interfaccia comune (*View*) per aggiornare i sistemi di visualizzazione, ed una classe (**ViewList**) che mantiene l'elenco dei sistemi di visualizzazione che devono essere notificati dei cambiamenti di stato. Le classi **TextDisplay** e **GraphicDisplay** realizzano l'interfaccia di *View*. Un'applicazione che deve gestire l'insieme di dati può creare una

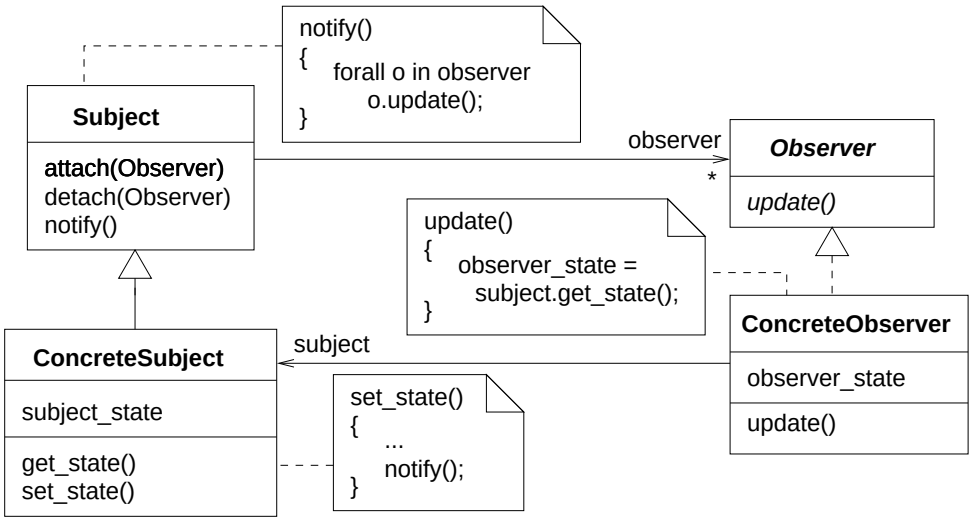


Figura 6.48: Design pattern *Observer*.

o piú istanze delle due classi ed aggiungerla alla lista di oggetti *View* mantenuta da **ViewList**. Ogni volta che i dati vengono aggiornati con l'operazione set_data, questa usa l'operazione notify che invoca update sugli oggetti registrati nella lista. Ciascuno di questi, infine, visualizza i dati eseguendo update.

Il pattern corrispondente a questo schema di soluzione è mostrato in fig. 6.48.

6.5.9 Singleton

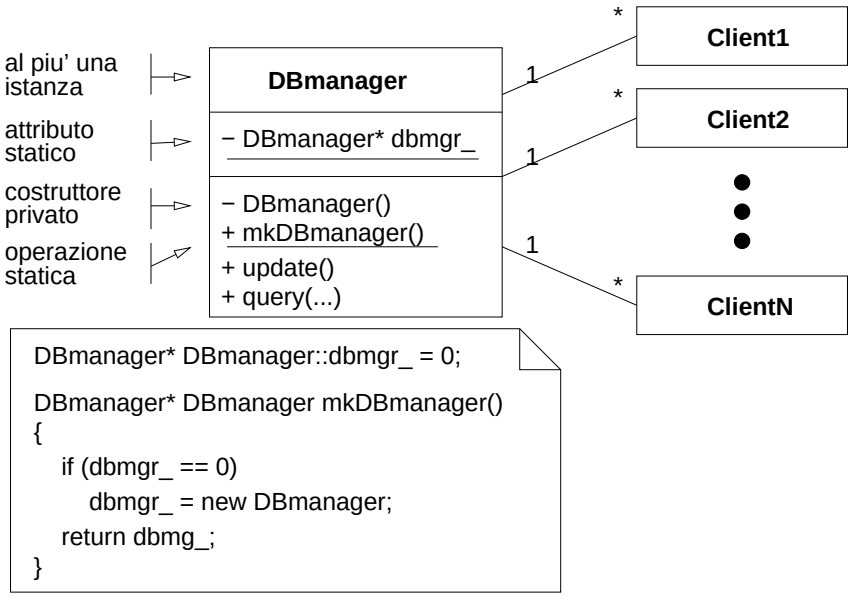


Figura 6.49: Problema: avere una sola istanza di una classe.

Il pattern Singleton si usa per garantire che nel sistema esista una sola istanza di una certa classe.

Per esempio, supponiamo che un un certo numero di applicazioni debba accedere ad un'istanza della classe **DBmanager**, che gestisce un database (fig. 6.49).

Per evitare che si possano creare piú istanze di questa classe, si assegna la visibilità privata (o protetta) al suo costruttore. Le applicazioni clienti ottengono un puntatore all'unica istanza per mezzo di un'operazione statica. Questa alloca un'istanza del gestore in memoria dinamica solo la prima volta che viene invocata.

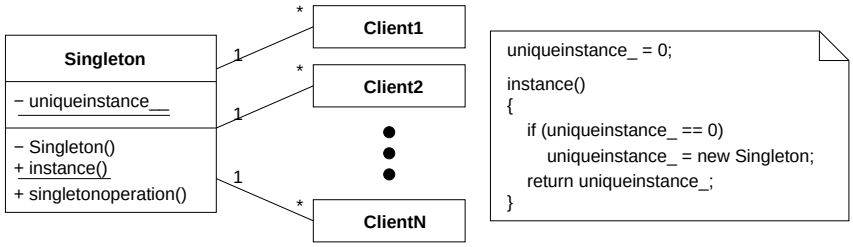


Figura 6.50: Design pattern *Singleton*.

La fig. 6.50 mostra la struttura del pattern.

6.6 Progetto dettagliato

In questa sezione consideriamo alcune linee guida per la fase di progetto in dettaglio, che sfuma nella fase di codifica. In questa fase vengono precisati i dettagli delle singole classi e associazioni, e vengono introdotte classi ed associazioni ausiliarie.

6.6.1 Progetto delle classi

Nella fase di progetto architetturale le classi definite nel modello di analisi vengono raggruppate in componenti insieme a qualche nuova classe introdotta in questa fase. Generalmente queste classi non sono state definite completamente, quindi nel progetto dettagliato bisogna arrivare prima di tutto al completamento della loro definizione. Per questo occorre:

- specificare completamente attributi ed operazioni già presenti, indicando visibilità, modificabilità, tipo e direzione dei parametri;
- aggiungere operazioni implicite nel modello, per esempio costruttori e distruttori;
- aggiungere operazioni ausiliarie, se necessario.

Oltre a completare e raffinare le definizioni delle singole classi, si possono fare delle operazioni di ristrutturazione, per esempio scomponendo una classe in classi piú piccole, oppure riunendo piú classi in una, o ridistribuendo fra piú classi le rispettive operazioni,

sempre cercando la massima coesione entro ciascuna classe. È possibile anche riorganizzare le gerarchie di generalizzazione. L'obiettivo di questi procedimenti è di far sí che ogni classe goda delle seguenti proprietà [8]:

coesività: la classe deve avere una responsabilità ben delimitata, e, se necessario, delegare ad altre classi l'esecuzione di compiti richiesti da tale responsabilità.

completezza: la classe deve offrire tutte le operazioni che i progettisti delle classi clienti si possono aspettare, in base alla specifica della classe ed anche al suo nome (scegliere nomi descrittivi ma sintetici, e non fuorvianti).

sufficienza: la classe deve offrire solo le operazioni richieste dalla specifica; questa proprietà è legata anche a quella di coesività.

primitività: ciascuna operazione deve implementare un'unica funzione elementare: se un certo scopo può essere raggiunto in piú modi, si deve scegliere il piú fondamentale e generalmente applicabile ed implementarlo con una operazione *primitiva* con cui si possano implementare modi alternativi, se e quando necessario.

disaccoppiamento: la classe dev'essere associata al minor numero possibile di classi.

6.6.2 Progetto delle associazioni

Le associazioni rappresentano i percorsi logici attraverso cui si propagano le interazioni fra i vari oggetti. Nel progetto delle associazioni si cerca di ottimizzare tali percorsi, ristrutturandoli se necessario, per esempio aggiungendo associazioni ausiliarie che permettono un accesso piú efficiente, o anche eliminando associazioni ridondanti.

Le associazioni devono poi essere ristrutturate in modo da permettere la loro implementazione nei linguaggi di programmazione. Questi, infatti, non hanno dei concetti primitivi che corrispondano direttamente alle associazioni, che devono quindi essere tradotte in concetti a piú basso livello. A questo scopo bisogna prima di tutto che le associazioni definite nel modello di analisi (ed eventualmente nel modello architetturale) vengano specificate completamente, indicandone le molteplicità, i nomi dei ruoli e la *navigabilità*. Quest'ultima proprietà è la possibilità di accedere dalle istanze di una classe a quelle di un'altra: un'associazione fra le classi **A** e **B** è *navigabile da* ***A*** *(****B****) a* ***B*** *(****A****)* se ogni istanza di **A** (**B**) contiene le informazioni (p.es., un puntatore) per accedere a istanze di **B** (**A**). La navigabilità in una sola direzione si rappresenta con una punta di freccia all'estremo dell'associazione. Un'associazione senza frecce è navigabile nei due versi.

Avendo precisato queste informazioni, si possono considerare i casi (fig. 6.51) trattati nel séguito.

Associazioni da uno a uno e da molti a uno

In questo caso l'associazione può essere implementata come un semplice puntatore (e in questo caso si lascia immutata nel modello) o anche come un attributo, se la classe associata è molto semplice (p.es., `string`) oppure se si vuole modellare una composizione.

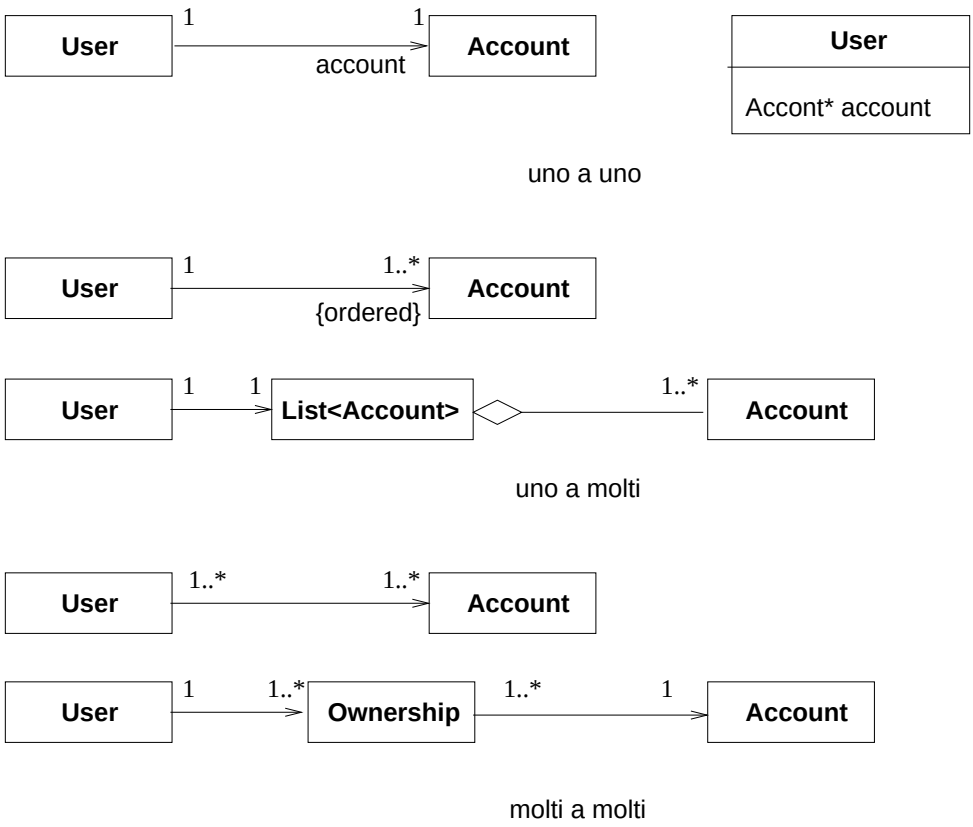


Figura 6.51: Realizzazione delle associazioni.

Associazioni da uno a molti

Nell'associazione da uno a molti un'istanza della classe all'origine dell'associazione è associata ad un gruppo di istanze dell'altra classe. In questo caso conviene introdurre una classe contenitore, possibilmente scelta fra le numerose classi (o template) di libreria disponibili per i vari linguaggi. La scelta della classe contenitore dipende dalle proprietà del gruppo di istanze, in particolare quelle di *unicità* (se ogni elemento appare una sola volta o no) e dell'*ordinamento*. Queste proprietà si possono rappresentare con le proprietà UML unique, nonunique, ordered, unordered. Per modellare le associazioni si può scegliere di lasciarle come sono, con l'indicazione di queste proprietà che serve da guida per il programmatore nella scelta della classe contenitore, oppure rappresentare questa classe esplicitamente.

Associazioni da molti a molti

Questo tipo di associazione si realizza introducendo una classe intermedia, come in fig. 6.51, e risolvendo le associazioni risultanti come nei casi precedenti.

Letture

Obbligatorie: cap. 4 Ghezzi, Jazayeri, Mandrioli. Pattern Composite, Adapter, Bridge, Proxy, Abstract Factory, Iterator su Gamma *et al.*

Facoltative: Dispense sul sito del corso[13].

[13] `http://www.ing.unipi.it/~a009435/issw/esercitazioni/1213/design_ptns/`

Capitolo 7

Convalida e verifica

Ogni attività industriale richiede che vengano controllate la correttezza, la funzionalità e la qualità dei prodotti finiti, dei prodotti intermedi e dello stesso processo di produzione. Come spiegato nella sez. 3.1.7, nell'attività di convalida i prodotti vengono confrontati con i requisiti dell'utente, mentre nella verifica il confronto avviene con le specifiche. Ricordiamo che i requisiti sono generalmente imprecisi e necessariamente informali, mentre le specifiche (risultanti dalla fase di analisi dei requisiti) sono precise e possono essere formali. La verifica può quindi contare su procedimenti piú metodici e rigorosi, però non è sufficiente ad assicurare la bontà del prodotto, poiché le specifiche stesse, che ne sono il punto di riferimento, possono non essere corrette rispetto ai requisiti e devono essere a loro volta convalidate. Verifica e convalida devono quindi completarsi a vicenda. Aforisticamente, si dice che con la verifica ci accertiamo che il prodotto sia fatto bene (*building the product right*) e con la convalida ci accertiamo che il prodotto sia quello giusto (*building the right product*).

Nel séguito, faremo riferimento principalmente alla verifica (e marginalmente alla convalida) del codice, pur tenendo presente che, come già accennato, i prodotti di ogni fase del processo di sviluppo devono essere verificati o convalidati.

7.1 Concetti fondamentali

Introduciamo qui alcuni concetti fondamentali nel campo della convalida e della verifica, cominciando dai termini *errore*, *anomalia*, e *guasto*[1]:

guasto: (*malfunzionamento*, *failure*) un comportamento del programma non corretto rispetto alle specifiche.
anomalia: (*difetto*, *fault*, *bug*) un aspetto del codice sorgente che provoca dei guasti.
errore: errore concettuale o materiale che causa anomalie.

[1] Questi termini hanno significati diversi nel campo della tolleranza ai guasti.

Gli errori sono quindi la causa delle anomalie, che a loro volta sono la causa dei malfunzionamenti. Osserviamo però che non c'è una corrispondenza diretta e biunivoca fra anomalie e guasti. Un particolare guasto può essere provocato da una o piú anomalie, e l'effetto di una anomalia può essere bilanciato, e quindi nascosto, da quello di un'altra. Alcune anomalie possono non provocare alcun guasto. Ma soprattutto, il problema fondamentale è che in genere un malfunzionamento avviene in presenza di alcuni dati di ingresso o di alcune *sequenze* di dati di ingresso, e non di altri. Nei sistemi in tempo reale, inoltre, i malfunzionamenti si possono verificare o no, a seconda dell'istante in cui i dati (o gli stimoli) vengono presentati all'ingresso del sistema.

Le attività rivolte alla ricerca e all'eliminazione delle anomalie si possono classificare come segue:

analisi statica: esame del codice sorgente.
analisi dinamica: esecuzione del codice.
debugging: ricerca delle anomalie a partire dai guasti, e loro eliminazione.

Nell'analisi dinamica, in senso piú ampio, possono rientrare l'*esecuzione di prototipi* e la *simulazione*. Quest'ultima permette di eseguire il codice entro un ambiente che simula il sistema complessivo in dovrà operare.

Conviene inoltre distinguere fra convalida o verifica *in piccolo* e *in grande*, cioè a livello di singolo modulo (o *unità*) o di sistema.

7.2 Analisi statica

L'analisi statica consiste nell'esame del codice sorgente o piú in generale di documenti di specifica o di progetto. Fra i vari tipi di analisi statica citiamo i seguenti:

walk-through: un gruppo di collaudatori, progettisti ed eventualmente rappresentanti del committente o degli utenti analizza il codice e ne simula l'esecuzione con carta e penna.

ispezione: simile al walk-through, ma finalizzata alla scoperta di specifici errori (p.es., uso scorretto dei puntatori o della memoria dinamica, variabili non inizializzate, ...).

analisi automatica: uso di strumenti (inclusi i compilatori) per cercare possibili sintomi di anomalie, come istruzioni non raggiungibili o variabili non inizializzate.

esecuzione simbolica: si assegnano valori simbolici (per esempio, x) alle variabili d'ingresso, ed un *interprete simbolico* esegue il programma calcolando i valori di uscita sotto forma di espressioni algebriche che si possono confrontare con le specifiche.

formalizzazione: traduzione di un documento in un linguaggio formale, in modo da convalidarlo e costruire un modello per la verifica formale.

verifica formale: dimostrazione formale di correttezza, generalmente svolta sulle linee dell'esempio nella sez. 4.5.4, specificando pre- e postcondizioni di un segmento di codice e dimostrando che le postcondizioni sono conseguenza delle precondizioni e dell'esecuzione del codice.

7.3 Analisi dinamica

> *Program testing can be used to show the* presence *of bugs, but never to show their absence.*
> – E.W. Dijkstra, citato in Dahl et al., *Structured Programming.*

Nell'analisi dinamica (o *testing*) si esegue il programma da verificare o convalidare per osservare se i risultati corrispondono a quelli previsti dalle specifiche o dai requisiti. Generalmente è impossibile eseguire il programma per tutti i valori possibili dei dati di ingresso e per tutte le possibili condizioni al contorno (p.es., presenza di determinate risorse, diversi ambienti di esecuzione, ...), poiché questo insieme può essere infinito o comunque troppo grande. Il primo problema da affrontare nell'attività di testing è quindi quello di selezionare un insieme di dati di ingresso che sia abbastanza piccolo da permettere l'esecuzione dei test entro i limiti di tempo e di risorse disponibili, ma sia anche abbastanza grande, e soprattutto significativo, da fornire risultati sufficientemente affidabili. Il secondo problema è quello di stabilire il criterio di successo o fallimento del test, cioè capire quale deve essere il comportamento corretto del programma. Infine, bisogna stabilire il criterio di terminazione, per decidere quando concludere l'attività di testing.

Riassumiamo qui, schematicamente, i problemi appena citati:

selezione dei dati: può essere guidata:

- dalle specifiche (test funzionale);
- dalla struttura del programma (test strutturale);

nessuno dei due criteri da solo è sufficiente, poiché ciascuno di essi permette di scoprire tipi diversi di guasti.

correttezza dei risultati: può essere decisa:

- dall'utente (convalida);
- dal confronto con le specifiche (verifica); in questo caso, le specifiche si rivelano tanto piú utili quanto piú sono formali; se le specifiche sono eseguibili, il confronto può avvenire con i risultati di un prototipo ottenuto direttamente dalle specifiche;
- dal confronto con versioni precedenti; avviene nei test di *regressione*, in cui si verifica una nuova versione del software.

terminazione: può essere decisa in base a modelli statistici che permettano di stimare il numero di anomalie sopravvissute ai test, oppure in base a criteri di copertura (sez. 7.3.1).

È fondamentale, infine, tener presente la già citata

Tesi di Dijkstra Un test può rilevare la presenza di malfunzionamenti, ma non dimostrarne l'assenza.

7.3.1 Test strutturale

Nel test strutturale la selezione dei dati di test viene guidata dalle informazioni che abbiamo sul codice da testare, e presuppone quindi che il codice sorgente sia disponibile. L'idea generale è di trovare dei dati di ingresso che causino l'esecuzione di tutte le operazioni previste dal programma, nelle varie sequenze possibili. Il problema fondamentale sta nel fatto che il numero delle possibili sequenze di operazioni in generale non è limitato: si pensi ai programmi contenenti cicli, che possono essere iterati per un numero di volte determinabile solo a tempo di esecuzione.

I dati di test vengono scelti in base a dei *criteri di copertura*, che definiscono l'insieme di sequenze di operazioni che devono essere eseguite (o *esercitate*, *exercised*) nel corso del test.

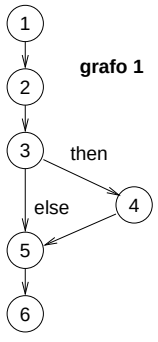
Il grafo di controllo

Nel test strutturale ci si riferisce ad una rappresentazione astratta del programma, detta *grafo di controllo*. Nella sua forma piú semplice, il grafo di controllo viene costruito associando un nodo a ciascuna istruzione o condizione di istruzioni condizionali o iterative, e collegando i nodi con archi in modo tale che i cammini del grafo rappresentino le possibili sequenze di esecuzione. A seconda del grado di dettaglio desiderato, piú istruzioni possono essere rappresentate da un solo nodo, oppure una istruzione può essere scomposta in operazioni elementari, ciascuna rappresentata da un nodo.

Criterio di copertura dei comandi

Questo criterio richiede che ogni istruzione eseguibile del programma (cioè ogni nodo del grafo) sia eseguita per almeno un dato appartenente al test. Per esempio, consideriamo il seguente programma col suo grafo di controllo:

```
main()
{
    int x, y;                  // 1
    cin >> x >> y;             // 2
    if (x != 0)                // 3
        x = x + 10;            // 4
    y = y / x;                 // 5
    cout << x << endl << y; // 6
}
```

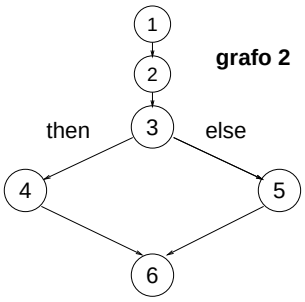


Il dominio del programma è l'insieme delle coppie ordinate di interi. Qualsiasi test contenente coppie (x, y) con x diverso da zero soddisfa il criterio di copertura. Osserviamo però che questo criterio esclude i test contenenti coppie con x nullo, in corrispondenza delle quali si verifica un malfunzionamento.

Criterio di copertura delle decisioni

Il criterio di copertura delle decisioni richiede che ogni arco del grafo di controllo venga percorso almeno una volta. Per il programma visto precedentemente, il criterio di copertura delle decisioni richiede che il test contenga coppie sia con x diverso da zero che con x uguale a zero, e in questo modo si trova il malfunzionamento dovuto alla divisione per zero. Ma per altri programmi il questo criterio non basta. Consideriamo quest'altro programma:

```
main()
{
    int x, y;                 // 1
    cin >> x >> y;            // 2
    if (x == 0 || y > 0)      // 3
        y = y / x;            // 4
    else
        x = y + 2 / x;        // 5
    cout << x << endl << y; // 6
}
```



Per il criterio di copertura delle decisioni, ogni test deve contenere sia coppie tali che la condizione `(x = 0 || y > 0)` sia vera, sia coppie tali che la condizione sia falsa. Un test accettabile è `{(x = 5, y = 5), (x = 5, y = -5)}`. Nemmeno questo test è capace di rilevare il malfunzionamento dovuto alla divisione per zero.

Criterio di copertura delle condizioni

Il problema dell'esempio precedente sta nel fatto che nel programma la decisione se eseguire il ramo `then` o il ramo `else` dipende da una condizione logica complessa. Il criterio di copertura delle condizioni è piú fine del criterio di copertura delle decisioni, poiché richiede che ciascuna condizione elementare venga resa sia vera che falsa dai diversi dati appartenenti al test. Nell'esempio considerato, questo criterio viene soddisfatto, per esempio, dal test `{(x = 0, y = -5), (x = 5, y = 5)}`.

Questo test, però, non soddisfa il criterio di copertura delle decisioni (si può verificare che per il test visto la condizione complessiva sia sempre vera), per cui non riesce a rilevare la divisione per zero nel ramo `else`.

Criterio di copertura delle decisioni e delle condizioni

Questo criterio combina i due precedenti, richiedendo che sia le condizioni elementari sia le condizioni complessive siano vere e false per diversi dati di test. Nell'esempio considerato, questo criterio viene soddisfatto dal test `{(x = 0, y = -5), (x = 5, y = 5), (x = 5, y = -5)}`; osserviamo però che in questo caso non si rileva la divisione per zero nel ramo `then`.

Questa tabella riassume i tre esempi relativi al grafo 2:

criterio soddisfatto	test set	condizioni x = 0	condizioni $y > 0$	ramo	trova guasto nel ramo
D	(x = 5, y = 5)	F	T	then	no
	(x = 5, y = -5)	F	F	else	no
C	(x = 5, y = 5)	F	T	then	no
	(x = 0, y = -5)	T	F	then	sí
DC	(x = 5, y = 5)	F	T	then	no
	(x = 0, y = -5)	T	F	then	sí
	(x = 5, y = -5)	F	F	else	no

D: decisioni, C: condizioni, DC: decisioni e condizioni

Criteri *data flow*

Nei criteri basati sul flusso dei dati, si cerca di coprire i percorsi contenenti operazioni che coinvolgono i valori delle variabili, come l'*assegnamento* (o *definizione*) e l'*uso* nelle istruzioni di calcolo o di controllo.

7.3.2 Test funzionale

Nel test funzionale bisogna prima di tutto analizzare un modello concettuale del sistema da collaudare, con l'obiettivo di trovare dei sottoinsiemi del dominio di ingresso da cui estrarre i dati di test.

Il modello si basa sui requisiti e sulle specifiche. Se queste ultime sono abbastanza precise, esse stesse costituiscono un modello da cui estrarre le informazioni necessarie per il test. Se le specifiche sono informali, devono essere analizzate per costruire il modello.

Classi di equivalenza e valori al contorno

Lo spazio dei valori dei dati d'ingresso si può ripartire in *classi di equivalenza*, cioè regioni in cui il comportamento del sistema è simile per tutti i valori appartenenti ad una data regione. I dati di test si scelgono quindi fra i valori interni a ciascuna classe e fra quelli appartenenti al suo contorno, ovvero al confine con altre classi. Questi ultimi sono i valori per cui in pratica ci si aspetta una maggiore probabilità di attivare guasti.

Supponiamo per esempio che un programma sia stato cosí specificato:

L'operazione A deve essere eseguita nel caso che la pressione sia minore o uguale a 3 bar, oppure la temperatura sia maggiore di 5 °C. *Altrimenti deve essere eseguita l'operazione B.*

Le classi di equivalenza sono costituite dagli insiemi $\{p \mid p \leq 3\}$ e $\{t \mid t > 5\}$, dalla loro unione, dalla loro intersezione e dal complemento della loro unione, come mostrato nel diagramma di fig. 7.1.

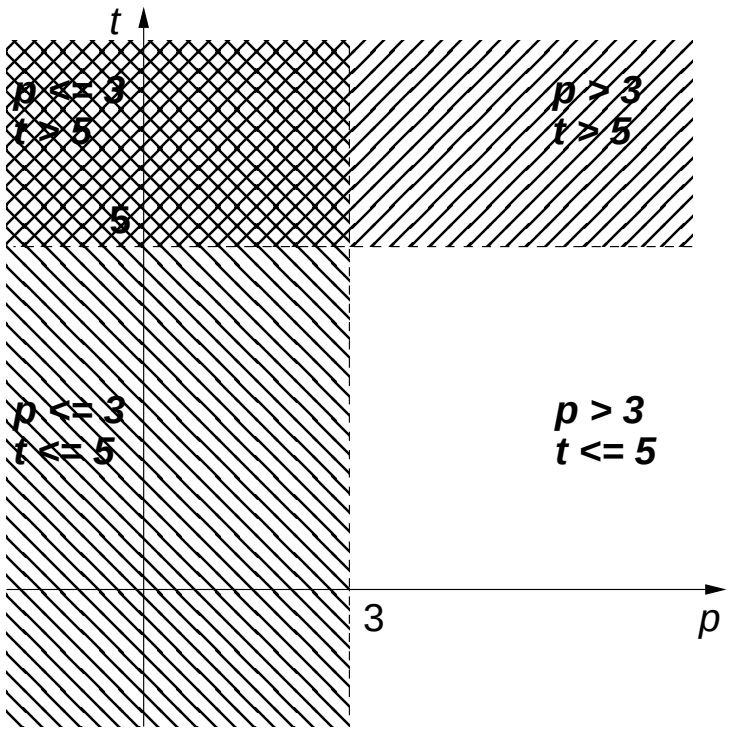


Figura 7.1: classi di equivalenza.

Tabelle di decisione e grafi causa/effetto

Le condizioni sui dati di ingresso e su quelli di uscita vengono rappresentate da variabili booleane. Le dipendenze logiche (AND, OR, NOT, XOR) possono essere rappresentate in forma tabulare o grafica, ma anche quando si usa la forma grafica, generalmente questa viene ricondotta ad una forma tabulare, piú adatta ad un'analisi sistematica e piú facilmente automatizzabile.

In forma tabulare, si usano *tabelle di decisione*. In queste tabelle le righe rappresentano i possibili valori booleani delle condizioni sui dati di ingresso e di uscita, e le colonne rappresentano le possibili combinazioni delle condizioni. I test vengono scelti in modo tale da esercitare le varie combinazioni degli ingressi.

Consideriamo, per esempio, un'interfaccia per un database, che accetti dei comandi per mostrare il contenuto di un indice, diviso in dieci sezioni (esempio tratto da [16]). Per chiedere la visualizzazione di una sezione, l'utente deve dare un comando formato da due caratteri, dei quali il primo è `D` oppure (indifferentemente) `L`, ed il secondo è la cifra corrispondente alla sezione richiesta (da 0 a 9). Se il comando è formato correttamente il sistema risponde visualizzando la sezione, se il primo carattere non è corretto il sistema

stampa un messaggio di errore (messaggio A), e se il secondo carattere non è corretto viene stampato un altro messaggio di errore (messaggio B). Per scrivere la tabella di decisione, dobbiamo individuare le condizioni booleane sugli ingressi (cause) e quelle sulle uscite (effetti), e numerarle per comodità di riferimento (i numeri sono stati scelti in modo da distinguere condizioni sugli ingressi, sulle uscite, e intermedie):

Cause	
1	il primo carattere è `D`
2	il primo carattere è `L`
3	il secondo carattere è una cifra
20	il primo carattere è `D` oppure `L`

Effetti	
50	visualizzazione della sezione richiesta
51	messaggio A
52	messaggio B

La causa 20 è intermedia fra le condizioni sugli ingressi e quelle sulle uscite. La tabella di decisione è la seguente:

Cause	combinazioni					
1	1	1	0	0	0	0
2	0	0	0	0	1	1
3	0	1	0	1	0	1
20	1	1	0	0	1	1
Effetti						
50	0	1	0	0	0	1
51	0	0	1	1	0	0
52	1	0	1	0	1	0

La prima colonna, per esempio, rappresenta la situazione in cui la condizione 1 è vera, le condizioni 2 e 3 sono false, e conseguentemente le condizioni 50 e 51 sono false e la condizione 52 è vera.

In forma grafica, si usano *grafi causa/effetto* i cui nodi terminali sono condizioni sui dati di ingresso (cause) e sui dati di uscita (effetti), ed i nodi interni sono condizioni intermedie, ciascuna espressa come combinazione logica semplice (AND, OR, NOT) delle rispettive cause. La fig. 7.2 mostra il grafo corrispondente all'esempio già esaminato.

Grafi

Un approccio generale al test funzionale si basa sulla costruzione di un grafo, al quale si possono applicare i criteri di copertura visti nel testing strutturale (ove il grafo considerato è il grafo di controllo del *codice*). Questo grafo si riferisce, a seconda dell'applicazione e/o del metodo di specifica, a vari aspetti *dell'applicazione* (e non dell'implementazione), quali il flusso del controllo, il flusso dei dati, l'evoluzione dello stato, ed altri ancora.

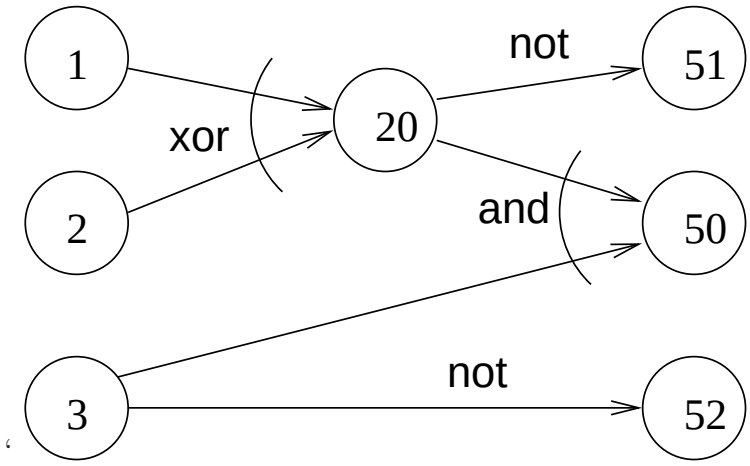


Figura 7.2: Grafo causa/effetto

7.4 CppUnit e Mockpp

Nel test di unità, il collaudo di ciascun modulo richiede l'uso di moduli ausiliari che simulano quelli che, nel sistema reale, interagiscono col modulo in esame. Questi moduli ausiliari devono simulare degli interi sottosistemi, ma ovviamente devono essere realizzati in modo semplice. Per esempio, invece di eseguire i calcoli richiesti, possono fornire al modulo sotto test i valori particolari richiesti dal test, ricavandoli da tabelle o dall'interazione con il collaudatore.

I framework CppUnit e Mockpp servono a facilitare la scrittura di programmi di test, sia nel test di unità che nel test di integrazione. La fig. 7.3 mostra l'architettura di un generico sistema in cui mettiamo in evidenza un componente da collaudare. Supponendo che prima di **UnderTest** sia stato implementato e collaudato solo il componente Provider1, bisogna realizzare l'infrastruttura di test (*test harness*) mostrata in fig. 7.4. I framework CppUnit e Mockpp servono a realizzare, rispettivamente, i moduli *driver* che simulano i clienti e i moduli *stub* che simulano i fornitori.

7.4.1 Il framework CppUnit

Per eseguire un test con questo framework, il collaudatore deve scrivere una classe driver. Ciascun metodo di questa classe esegue un caso di test, e il framework fornisce un programma che esegue i test e raccoglie i risultati. Per ogni caso di test, bisogna assicurare che valgano le precondizioni richieste, eseguire una o piú operazioni della classe sotto test, e infine verificare se valgono le postcondizioni specificate. In quest'ultima parte si usano asserzioni espresse con macro predefinite nel framework (`CPPUNIT_ASSERT()`).

Il framework CppUnit si può descrivere, in modo semplificato, come in fig. 7.5, dove **UnitA** è la classe da collaudare, **UnitB** è una classe che collabora con la precedente, e **UnitATest** è il driver. La classe **TestFixture** crea ed inizializza le istanze della classe

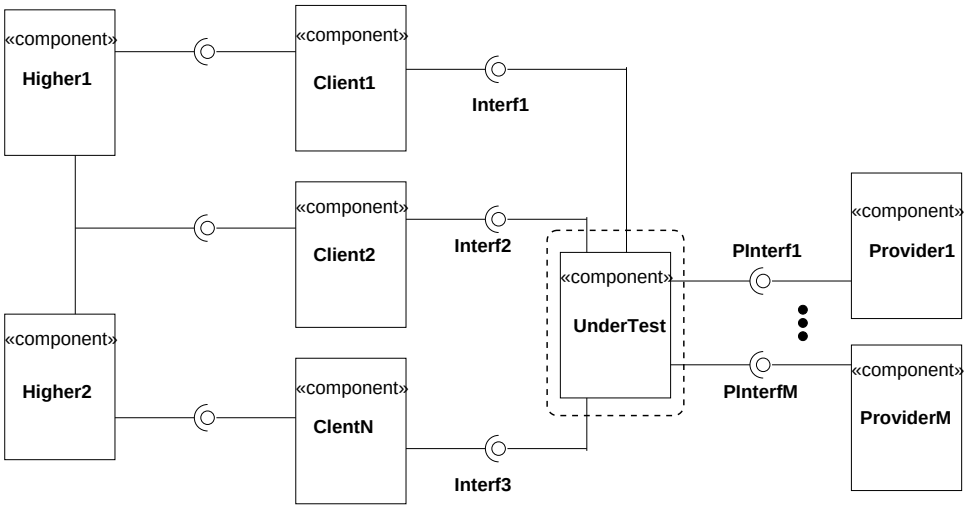


Figura 7.3: Integrazione di un componente.

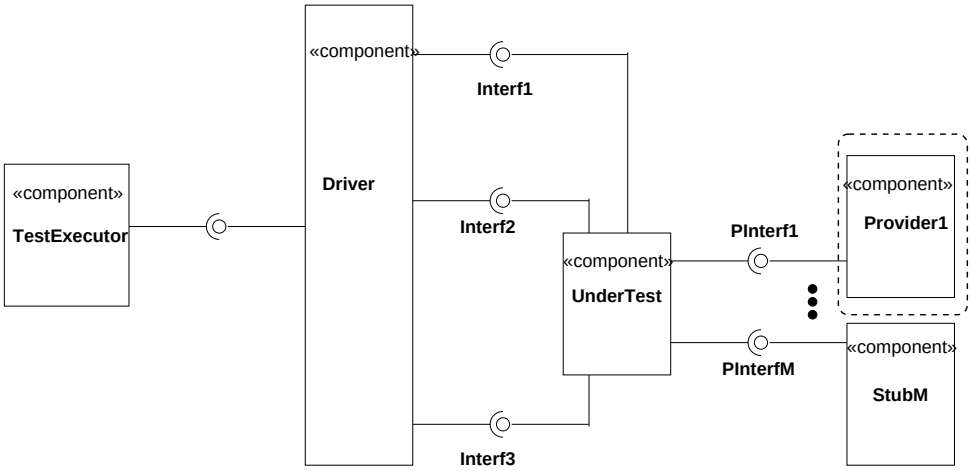


Figura 7.4: Infrastruttura di test.

sotto test e le altre classi, e poi le distrugge alla fine del test. **TestCaseAaCaller** esegue il caso di test *a*, cioè il metodo testCaseAa() della classe **UnitATest**, e **TestRunner** è la classe di piú alto livello nel programma di test, quella che coordina le altre.

La fig. 7.6 mostra l'applicazione del framework al test di una classe **Complex**, che supponiamo sia cosí definita:

```
class Complex {
    double real;
    double imaginary;
public:
    Complex(double r, double i = 0) : real(r), imaginary(i) {};
    friend bool operator==(const Complex& a, const Complex& b);
    friend Complex operator+(const Complex& a, const Complex& b);
};
```

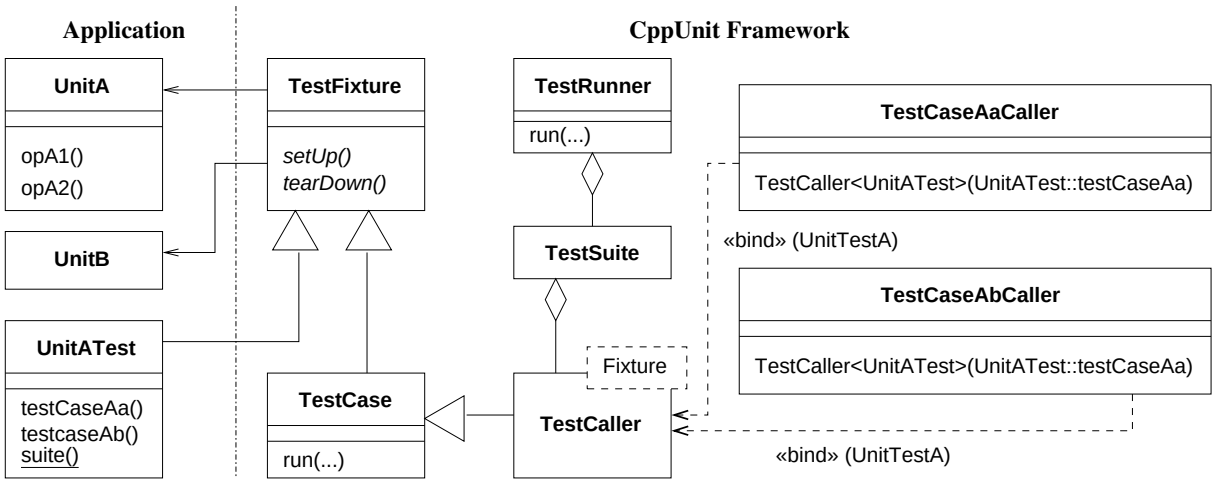


Figura 7.5: Infrastruttura CppUnit.

e cosí implementata:

```
bool
operator==(const Complex &a, const Complex &b)
{
    return a.real == b.real && a.imaginary == b.imaginary;
}

Complex
operator+(const Complex &a, const Complex &b)
{
    return Complex(a.real, a.imaginary + b.imaginary);
}
```

La classe driver è `ComplexTest`:

```
class ComplexTest : public CppUnit::TestFixture  {
private:
    Complex* m_10_1;
    Complex* m_1_1;
    Complex* m_11_2;
public:
    static CppUnit::Test *suite();
    void setUp();
    void tearDown() ;
    void testEquality();
    void testAddition();
};
```

dove `testEquality()` e `testAddition()` sono i casi di test. Gli altri metodi creano la sequenza di test da eseguire (`suite()`), creano le istanze di `Complex` da usare nei test (`setup()`), e le distruggono (`tearDown()`), come si vede nell'implementazione della classe:

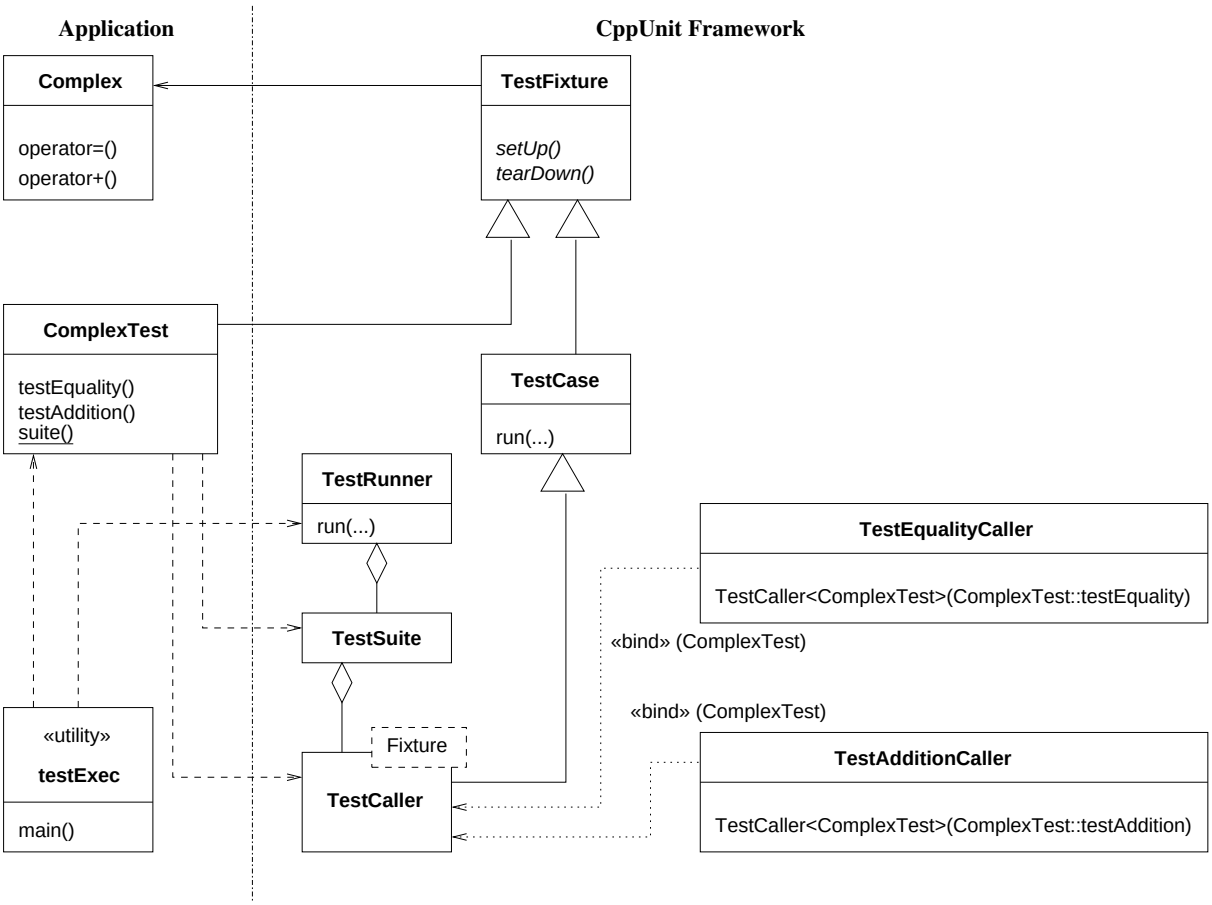


Figura 7.6: Infrastruttura di test per la classe **Complex**.

```
CppUnit::Test*
ComplexTest::
suite()
{
    CppUnit::TestSuite* suiteOfTests =
            new CppUnit::TestSuite("ComplexTest");
    suiteOfTests->addTest(new CppUnit::TestCaller<ComplexTest>(
                                    "testEquality",
                                    &ComplexTest::testEquality));
    suiteOfTests->addTest(new CppUnit::TestCaller<ComplexTest>(
                                    "testAddition",
                                    &ComplexTest::testAddition));
    return suiteOfTests;
}

void
ComplexTest::
setUp()
{
    m_10_1 = new Complex(10, 1);
    m_1_1 = new Complex(1, 1);
    m_11_2 = new Complex(11, 2);
}
```

```
void
ComplexTest::
tearDown()
{
    delete m_10_1;
    delete m_1_1;
    delete m_11_2;
}

void
ComplexTest::
testEquality()
{
    CPPUNIT_ASSERT(*m_10_1 == *m_10_1);
    CPPUNIT_ASSERT(!(*m_10_1 == *m_11_2));
}

void
ComplexTest::
testAddition()
{
    CPPUNIT_ASSERT(*m_10_1 + *m_1_1 == *m_11_2);
}
```

Il programma principale crea un istanza di (una sottoclasse di) `TestRunner`, la inizializza e la esegue:

```
int
main()
{
    CppUnit::TextTestRunner runner;
    runner.addTest(ComplexTest::suite());
    runner.run();
    return 0;
}
```

Le macro `CPPUNIT_ASSERT` nei casi di test verificano le condizioni specificate, e, se queste non risultano vere, scrivono dei messaggi di errore.

7.4.2 Il framework Mockpp

Il framework Mockpp si basa su una versione del pattern Adapter e si può descrivere come in fig. 7.7, dove **UnitA** è la classe da collaudare, **Provider1** è la classe richiesta da **UnitA** che dobbiamo simulare con uno stub, **IProvider1** è la sua interfaccia, e **MockProvider1** è lo stub. La classe **MockObject** è la base degli stub, le cui istanze possono contenere un insieme di istanze di classi derivate da **Expectation** o da **ReturnObject**. Le istanze di **Expectation** descrivono i valori che ci si aspetta vengano passati ai metodi di **Provider1** nell'esecuzione dei metodi di **UnitA**, mentre le istanze di **ReturnObject** contengono i valori che lo stub deve restituire al modulo sotto test.

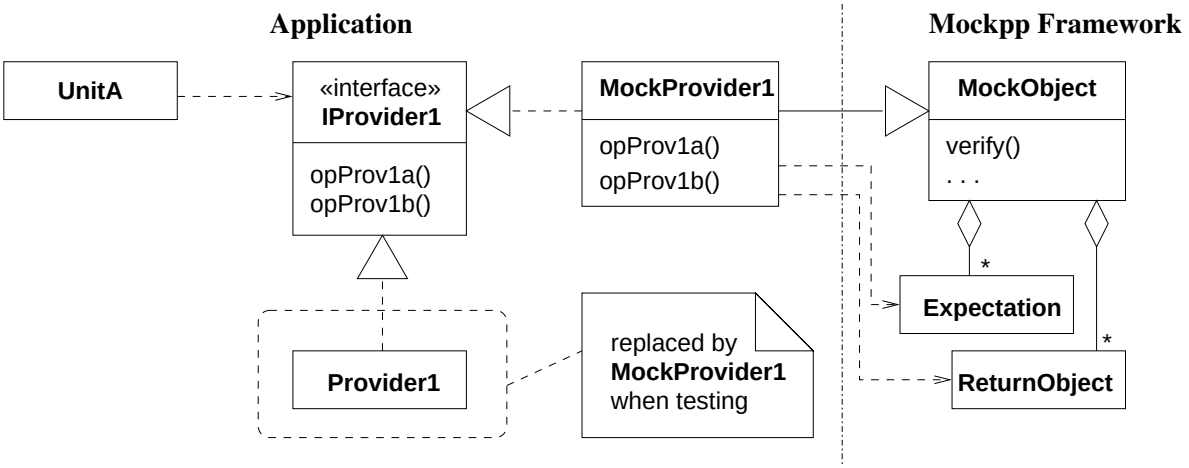


Figura 7.7: Infrastruttura Mockpp.

Le *expectation* definiscono il comportamento dello stub. Sono dei contenitori che, all'atto della creazione dello stub, vengono riempiti con valori o vincoli definiti dal collaudatore. Durante l'esecuzione del test le istanze dell'unità sotto test invocano i metodi dello stub, che passano i loro argomenti alle *expectation*. Queste ultime verificano se i valori degli argomenti corrispondono ai valori previsti o rispettano i vincoli specificati, sollevando un'eccezione se ci sono delle discrepanze. Le *expectation* possono eseguire diversi tipi di verifiche, come confrontare valori e controllare il loro ordinamento temporale.

La fig. 7.8 mostra un'applicazione di questo framework al collaudo di una classe **Consumer**. Una sua istanza apre un file di configurazione, legge tre righe e chiude il file, poi compie delle elaborazioni. Queste operazioni vengono eseguite da una classe (ancora da implementare) che realizzi l'interfaccia **Interface**. Vogliamo verificare che `Consumer` usi correttamente le operazioni di tale classe. Queste sono le dichiarazioni delle due classi:

```
class Interface
{
public:
    virtual void open(const std::string &name) = 0;
    virtual std::string read() = 0;
    virtual std write(const std::string &s) = 0;
    virtual unsigned calculate(unsigned input) = 0;
    virtual void close() = 0;
};

class Consumer {
    Interface* configfile;
    std::string config1, config2, config3;
public:
    Consumer(Interface* intf) : configfile(intf) {};
    void load();
```

```
    void process();
    void save();
};
```

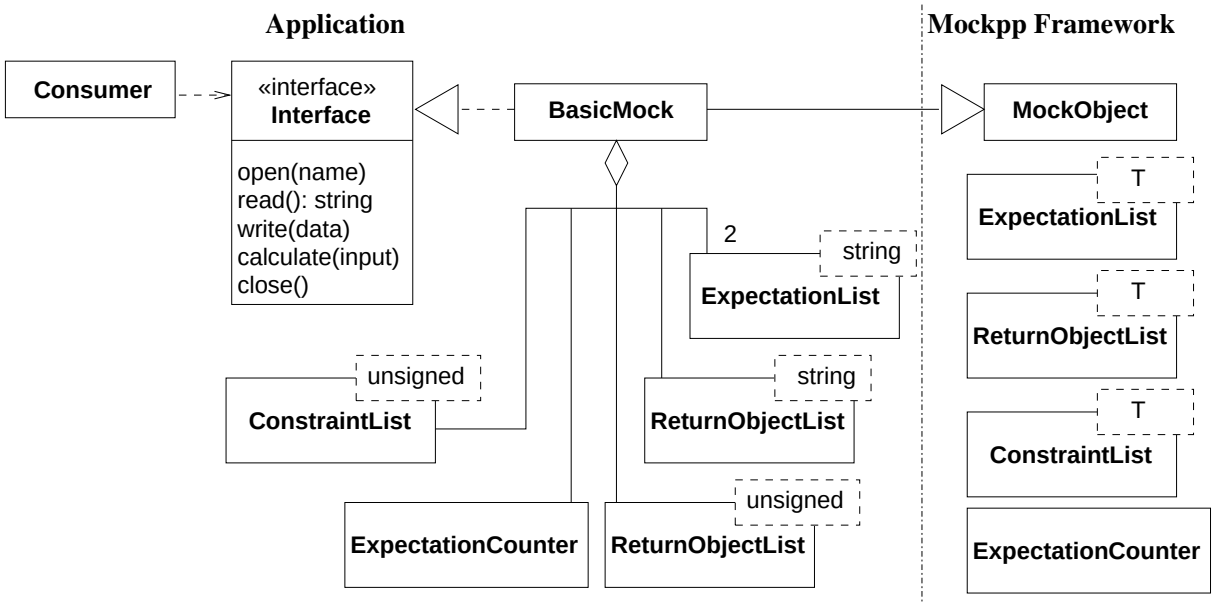


Figura 7.8: Infrastruttura Mockpp, esempio.

La classe **BasicMock** è lo stub che realizza **Interface** e deriva da **MockObject**:

```
class BasicMock : public Interface, public mockpp::MockObject {
```

I membri dato di questa classe sono istanze di classi di tipo expectation e return object:

```
  mockpp::ExpectationList<std::string>  open_name;
  mockpp::ExpectationList<std::string>  write_data;
  mockpp::ReturnObjectList<std::string> read_data;
  mockpp::ReturnObjectList<unsigned>    calculate_output;
  mockpp::ExpectationCounter            close_counter;
  mockpp::ConstraintList<unsigned >     calculate_input;
```

Le expectation di tipo `ExpectationList` contengono il nome del file da aprire (`open_name`) ed i caratteri da scrivere nel file (`write_data`), mentre quelle di tipo `ReturnObjectList` contengono i valori da restituire. L'expectation `close_counter`, di tipo `ExpectationCounter`, controlla che il file venga chiuso il numero previsto di volte. L'expectation `calculate_input` di tipo `ConstraintList` verifica che i dati passati alla procedura di calcolo rispettino i vincoli richiesti.

Le operazioni di `BasicMock` implementano `Interface`:

```
void
BasicMock::
open(const std::string& name)
{ open_name.addActual(name); }

std::string
BasicMock::
read()
{ return read_data.nextReturnObject(); }

void
BasicMock::
write(const std::string &s)
{ write_data.addActual(s); }

unsigned
BasicMock::
calculate(unsigned input)
{   calculate_input.addActual(input);
    return calculate_output.nextReturnObject();
}

void
BasicMock::
close()
{ close_counter.inc(); }
```

Possiamo osservare che le operazioni che devono fornire dati in ingresso allo stub sono implementate con chiamate `addActual()` su expectation di tipo `ExpectationList` o `ConstraintList`, mentre le operazioni che devono restituire valori sono implementate con operazioni `nextReturnObject()` su expectation di tipo `ReturnObjectList`. Per contare le invocazioni di un'operazione, si usa l'operazione `inc()` (incremento) su un'expectation di tipo `ExpectationCounter`.

Il programma principale deve prima di tutto istanziare lo stub ed inizializzare le varie expectation in base al test che si vuole eseguire. Il file si chiama `"file1.lst"` e deve essere aperto due volte, una in scrittura ed una in lettura, e conseguentemente verrà chiuso due volte:

```
int main(int argc, char **argv)
{ try {
    BasicMock mock;
    mock.open_name.addExpected("file1.lst");      // expctn
    mock.open_name.addExpected("file1.lst");      // expctn
    mock.close_counter.setExpected(2);
```

Quindi lo stub dovrà restituire delle stringhe di caratteri a tre operazioni di lettura:

```
    mock.read_data.addObjectToReturn("record-1"); // return
    mock.read_data.addObjectToReturn("record-2"); //   "
    mock.read_data.addObjectToReturn("record-3");
```

Poi l'unità sotto test dovrà passare altre stringhe allo stub:

```
    mock.write_data.addExpected("record-1/processed");
    mock.write_data.addExpected("record-2/processed");
    mock.write_data.addExpected("record-3/processed");
```

Infine, l'unità sotto test dovrà passare dei valori numerici allo stub, che ne restituirà altri. Sui valori passati allo stub è posto il vincolo che siano maggiori di 5, che si rappresenta con l'operatore `gt()` del framework:

```
    mock.calculate_input.addExpected(gt<unsigned>(5));
    mock.calculate_input.addExpected(gt<unsigned>(5));
    mock.calculate_input.addExpected(gt<unsigned>(5));
    mock.calculate_output.addObjectToReturn(10);
    mock.calculate_output.addObjectToReturn(20);
    mock.calculate_output.addObjectToReturn(30);
```

Dopo aver inizializzato le expectation, viene istanziata la classe sotto test e si eseguono le sue operazioni. Se qualche verifica fatta dallo stub fallisce, viene sollevata un'eccezione, altrimenti il programma termina con successo:

```
    Consumer consumer(&mock);                    // esecuzione
    consumer.load();                             //   "
    consumer.process();                          //   "
    consumer.save();                             //   "
    mock.verify();                               // controllo
    std::cout << "Test eseguito con successo" << std::endl;
  } catch(std::exception &ex) {
    std::cout << "Errori.\n" << ex.what() << std::endl
    return 1;
  }
  return 0;
}
```

7.5 Test in grande

I test in grande si applicano al sistema intero o a suoi sottosistemi e ne esistono varie categorie, di cui tratteremo molto brevemente solo le piú comuni.

7.5.1 Test di integrazione

Nel test di integrazione si vuole collaudare l'interfacciamento fra moduli il cui funzionamento è già stato testato individualmente. È possibile eseguire il test di integrazione dopo che tutti i moduli sono stati testati individualmente, assemblandoli e provando il sistema completo (*big-bang test*). In genere si preferisce una strategia incrementale del test di integrazione, che permette di integrare i moduli man mano che vengono completati e che superano il test di unità. In questo modo gli errori di interfacciamento vengono scoperti prima ed in modo piú localizzato, e si risparmia sul numero di driver e di stub che devono essere sviluppati.

La strategia del test di integrazione generalmente ricalca la strategia di sviluppo, e quindi può essere top-down o bottom-up, o una combinazione delle due. Con la strategia top down non c'è bisogno di driver, poiché i moduli sviluppati e testati in precedenza fanno da driver per i moduli integrati successivamente. Analogamente, con la strategia bottom-up non c'è bisogno di stub. La strategia top-down permette di avere presto dei prototipi, mentre la strategia bottom-up permette di testare subito i moduli di basso livello, che spesso sono i piú critici.

7.5.2 Test di sistema

Il test di integrazione, sebbene nella sua ultima fase si applichi al sistema completo, è rivolto a verificare la correttezza dei singoli moduli rispetto all'interfacciamento col resto del sistema. Quindi il test di integrazione non può verificare quelle proprietà globali del sistema che non si possono riferire a qualche particolare modulo o sottosistema. Alcune di queste proprietà sono, per esempio, la robustezza e la riservatezza (*security*).

Test di stress

Il test di stress consiste nel sottoporre il sistema ad uno "sforzo" superiore a quello previsto dalle specifiche, per assicurarsi che il superamento dei limiti non porti a malfunzionamenti incontrollati, ma solo ad una degradazione delle prestazioni. La grandezza che definisce lo sforzo dipende dall'applicazione: per esempio, potrebbe essere il numero di utenti collegati contemporaneamente ad un sistema multiutente, o il numero di transazioni al minuto per un database.

Test di robustezza

Il test di robustezza consiste nell'inserire dati di ingresso scorretti. Anche in questo caso ci si aspetta che il sistema reagisca in modo controllato, per esempio stampando dei messaggi di errore e, nel caso di sistemi interattivi, rimettendosi in attesa di ulteriori comandi dall'utente. Il test di robustezza è particolarmente legato ai requisiti di sicurezza e di tolleranza ai guasti.

Test di accettazione

Il test di accettazione è un test di sistema eseguito dal committente (invece che dal produttore), che in base al risultato decide se accettare o no il prodotto. Il test di accettazione ha quindi una notevole importanza legale ed economica, e la sua pianificazione può far parte del contratto.

Un tipo di test di accettazione è quello usato per prodotti destinati al mercato. In questo caso il test avviene in due fasi: nella prima fase (α-*test*) il prodotto viene usato all'interno dell'organizzazione produttrice, e nella seconda (β-*test*) viene distribuito in prova ad alcuni clienti selezionati.

Test di regressione

Il test di regressione serve a verificare che una nuova versione di un prodotto non produca nuovi errori rispetto alle versioni precedenti e sia compatibile con esse. Il test di regressione può trarre grande vantaggio da una buona gestione del testing, che permetta di riusare i dati ed i risultati dei test delle versioni precedenti.

7.6 Il linguaggio TTCN-3

Il linguaggio *Test and Testing Control Notation 3* (TTCN-3) è stato concepito espressamente per l'esecuzione di test, è orientato al test di sistemi reattivi, e viene usato soprattutto nel settore delle telecomunicazioni per eseguire il *test di conformità* (*conformance*), che verifica il rispetto di determinati standard, e in particolare dei protocolli di comunicazione. Il TTCN-3 è comunque applicabile a diversi tipi di test e di applicazioni. Il linguaggio permette di definire componenti che stimolano il sistema sotto test e verificano la correttezza dei messaggi di risposta. I componenti possono essere eseguiti in parallelo, permettendo cosí di verificare sistemi concorrenti e distribuiti.

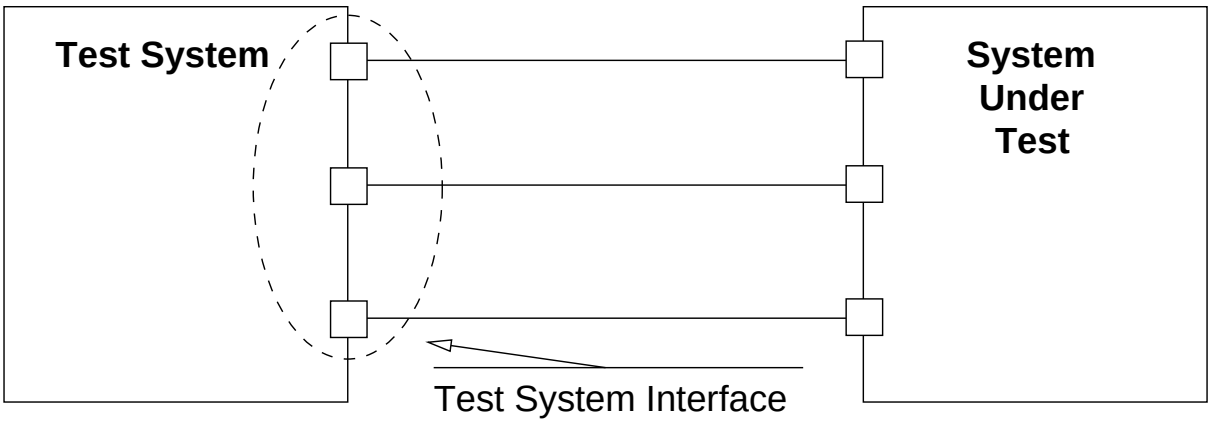


Figura 7.9: Sistema di test e sua interfaccia.

La fig. 7.9 mostra lo schema fondamentale di un'architettura di test basata su TTCN-3, e mette in evidenza l'interfaccia del sistema di test, definita da un insieme di *port* (analoghi a quelli visti in sez. 5.4.1), a loro volta caratterizzati dal tipo dei messaggi trasmessi e dalla loro direzione (*in*, *out*, *inout*). Il sistema da collaudare può essere un sistema software o un sistema fisico (p.es., una rete di comunicazione o un dispositivo).

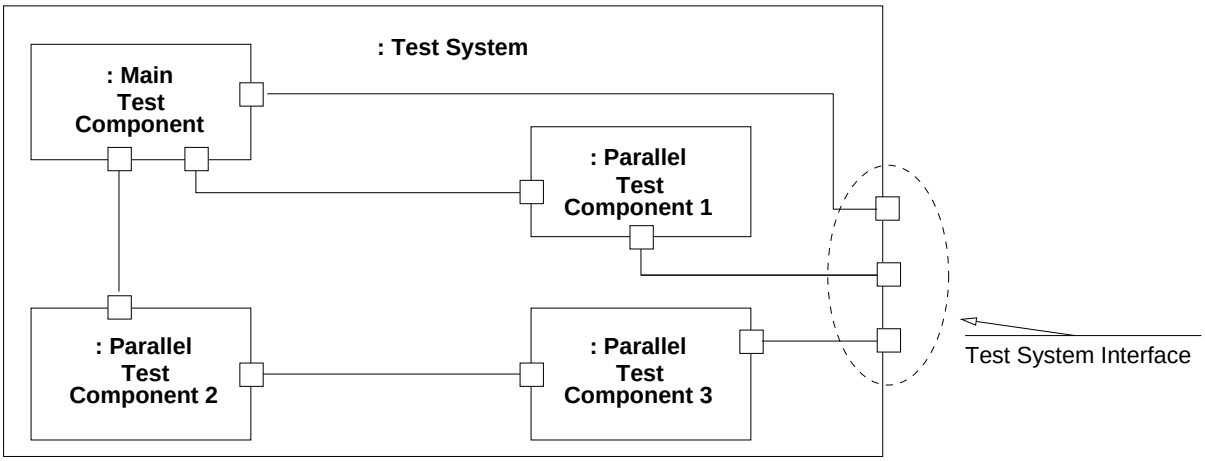


Figura 7.10: Componenti del sistema di test.

Per eseguire i test, il collaudatore scrive delle procedure `testcase`, che istanziano i componenti, li collegano fra di loro connettendo i port, e attivano i componenti. Un componente può contenere altri componenti, e i port del componente piú esterno (*sistema di test*) devono essere collegati al sistema sotto test. Questo collegamento avviene attraverso un *adattatore*, che è un componente scritto nel linguaggio del sistema sotto test, usando un framework che offre al sistema di test (scritto in TTCN-3) un'interfaccia predefinita. Il collaudatore deve reimplementare le operazioni di tale interfaccia in modo da adattarla al sistema sotto test.

Il sistema di test contiene un *Main Testing Component* (*MTC*) e, opzionalmente, dei *Parallel Testing Component* (*PTC*) (fig. 7.10). I collegamenti fra componenti interni al sistema di test vengono creati con clausole `connect`, quelli fra componenti interni e port dell'interfaccia esterno vengono creati con clausole `map`.

Alcune caratteristiche salienti del linguaggio si possono riassumere come segue:

- un sottoprogramma `testcase` è una procedura che esegue un test usando i port ed altre eventuali risorse dichiarate da un tipo di componente;
- la configurazione del sistema di test si può definire *dinamicamente* (cioè a tempo di esecuzione) mediante clausole `connect` e `map`;
- le comunicazioni fra componenti o fra componenti e test case possono essere *sincrone* (chiamate di procedura) o *asincrone* (messaggi);
- per le comunicazioni asincrone sono disponibili varie operazioni (`send`, `receive` e molte altre);
- flussi di controllo alternativi o concorrenti si possono rappresentare, rispettivamente, con le istruzioni `alt` ed `interleave`;

- il sistema di tipi è simile all'ASN.1, e include il tipo enumerato `verdict` per rappresentare l'esito dei test, che può essere *nessun risultato* (`none`), *test superato* (`pass`), *test fallito* (`fail`) e *risultato inconclusivo* (`inconc`);
- i formati dei messaggi si possono specificare usando `template` ed *espressioni regolari*;
- si possono creare e gestire dei `timer`.

Un'architettura di test è organizzata in *moduli*. Un modulo ha una parte *dichiarativa* ed una *di controllo* (opzionale). Nella parte dichiarativa si definiscono tipi di dati, tipi di port, tipi di componenti, template, casi di test, funzioni, eccetera. Nella parte di controllo si eseguono i casi di test, invocati con l'istruzione `execute`. Si possono usare istruzioni complesse come `if`, `for`, eccetera.

7.6.1 Esempio

```
(c) Copyright Wiley & Sons 2005

author: Colin Willcock, Thomas Deiß, Stephan Tobies,
         Stefan Keil, Federico Engler, Stephan Schulz
desc:   This is a strongly simplified Domain Name Server (DNS)
         test suite for testing some basic domain name
         resolution behaviour.
remark: This TTCN-3 code is based on the DNS example code
         presented in "C. Willock et al., An Introduction
         to TTCN-3, Wiley & Sons, 2005.  ISBN: 0-470-01224-2"
         This copyright notice shall not be removed in copies
         of this file.
```

Figura 7.11: Copyright per l'esempio "server DNS".

L'esempio seguente è tratto da materiale soggetto al copyright di fig. 7.11. Si deve collaudare un server DNS (*Domain Name System*) che riceve dai clienti delle *richieste* contenenti un nome di dominio e restituisce *risposte* contenenti il numero IP (*Internet Protocol*) corrispondente.

Il sistema di test è definito nel modulo `DNS` in cui viene dichiarato il tipo `DNSMessage`, che è un record con quattro campi, di cui uno opzionale:

```
module DNS {
  // tipi base
  type integer Identification( 0..65535 ); // 16-bit integer
  type enumerated MessageKind {e_Question, e_Answer};
  type charstring Question;
  type charstring Answer;
```

```
  // struttura generale dei messaggi
  type record DNSMessage {
    Identification identification,
    MessageKind messageKind,
    Question question,
    Answer answer optional      // campo opzionale
  }
```

Quindi vengono dichiarati i modelli per i messaggi di richiesta (`a_DNSQuestion`) e di risposta (`a_DNSAnswer`), che sono forme particolari del tipo `DNSMessage`, parametrizzate rispetto all'identificatore del messaggio e al contenuto dello stesso:

```
template DNSMessage        // modello dei msg di richiesta
   a_DNSQuestion(Identification p_id, Question e_question) :=
{ identification := p_id,
  messageKind := e_Question,
  question := e_question,  // campo per hostname
  answer := omit           // non c'e' il campo della risposta
}
template DNSMessage        // modello dei msg di risposta
   a_DNSAnswer(Identification p_id, Answer e_answer) :=
{ identification := p_id,
  messageKind := e_Answer,
  question := ?,           // qualsiasi valore di tipo Question
  answer := e_answer       // campo per indirizzo IP
}
```

Si possono quindi definire port e componenti:

```
type port DNSPort message {  // manda e riceve messaggi
    inout DNSQuery;          // asincroni di tipo
}                            // DNSQuery

type component DNSTester {   // un DNSTester ha
    port DNSPort P;          // un port di tipo DNSPort
}
```

Si definiscono un messaggio di richiesta e la risposta corrispondente, ed un test case:

```
var query := a_DNSQuestion(12345, "www.research.nokia.com");
var reply := a_DNSAnswer(12345, "172.21.56.98");
```

```
testcase tc_testcase()
        runs on DNSTester {                 // usa DNSTester
    timer t;
    P.send(query);                          // invia richiesta
    t.start(20.0);                          // imposta timeout
    alt {                                   // eventi alternativi
    [] P.receive(reply) setverdict(pass); // risp. giusta
    [] P.receive setverdict(fail);        //  " sbagliata
    [] t.timeout setverdict(inconc);      // non pervenuta
    }
    stop;
}
```

Il test case tratta i tre possibili esiti nel blocco `alt`.

La sezione di controllo del modulo esegue il test case:

```
 control {
    execute(tc_testcase());
  }
}   // end module DNS
```

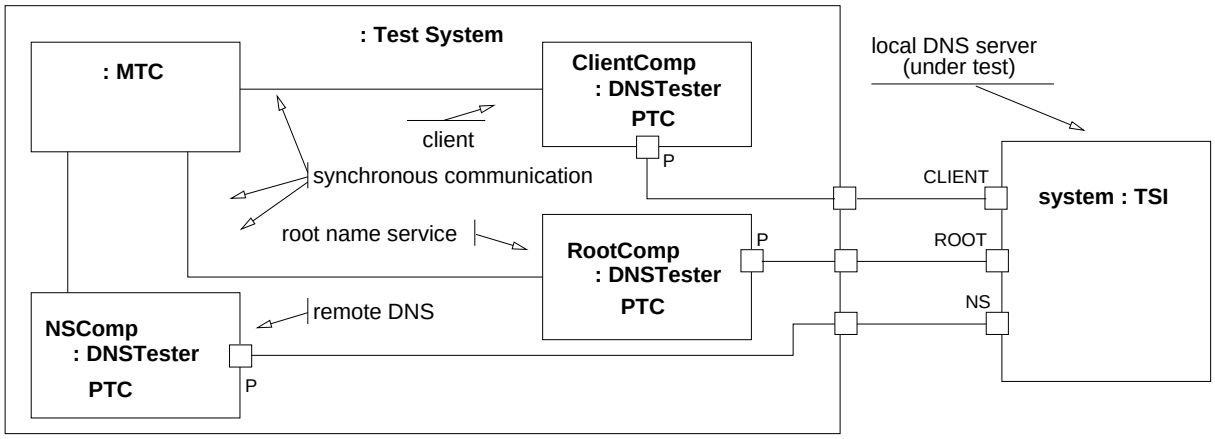


Figura 7.12: Sistema di test per il server DNS.

Possiamo estendere l'esempio usando componenti di test paralleli. Se un nome di domino non è nel database locale del server, la richiesta viene rediretta ad un name server remoto il cui indirizzo viene fornito da un server centrale (*root server*). La fig. 7.12 mostra l'architettura di test.

Al modulo già visto si aggiungono le definizioni dell'interfaccia del sistema di test, che ora comprende tre port, e del comportamento del root server:

```
type component TSI {  // test system interface
    port DNSPort CLIENT,
    port DNSPort ROOT,
    port DNSPort NS
}

function RootBehaviour() runs on DNSTester {
    alt {
    [] P.receive(rootquery) {
        P.send(rootanswer);
        setverdict(pass);
       }
    [] P.receive { setverdict(fail);}
    }
    stop;
}

function NSBehaviour() runs on DNSTester { ...  }

function ClientBehaviour() runs on DNSTester { ...  }
```

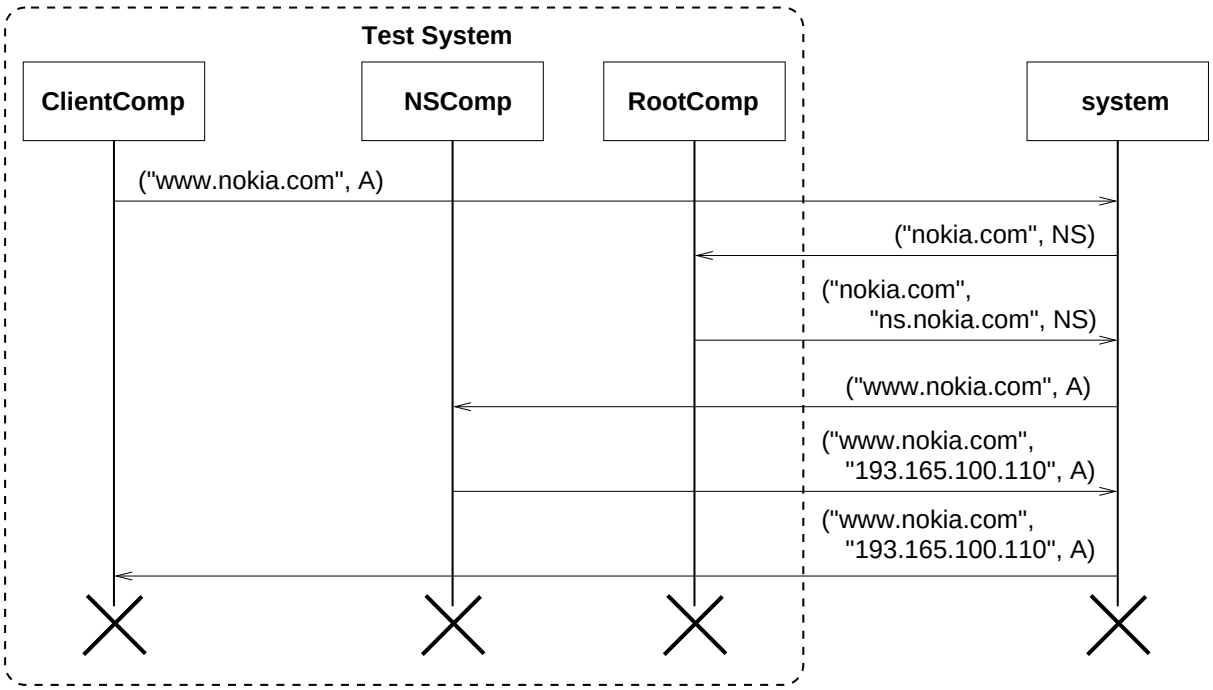


Figura 7.13: Esecuzione di un test case.

Si aggiunge poi un nuovo test case:

```
testcase Testcase3() runs on MTC system TSI {
```

```
  var DNSTester RootComp, NSComp, ClientComp;
  RootComp := DNSTester.create;     // crea componenti
  NSComp := DNSTester.create;       //             "
  ClientComp := DNSTester.create;   //             "
  map(RootComp:P, system:ROOT);     // crea collegamenti
  map(NSComp:P, system:NS);         //             "
  map(ClientComp:P, system:CLIENT); //             "
  RootComp.start(RootBehaviour());     // attivazione
  NSComp.start(NSBehaviour());         //             "
  ClientComp.start(ClientBehaviour()); //             "
  ClientComp.done;         // attesa fine di ClientComp
  stop;                             // fine del testcase
}
```

La fig. 7.13 mostra una possibile interazione.

Letture

Obbligatorie: cap. 6 Ghezzi, Jazayeri, Mandrioli [18], esclusa sez. 6.9, oppure cap. 13 e 14 Pressman [40].

Appendice A

Modello Cleanroom

Il modello Cleanroom [29] è un modello evolutivo concepito con l'obiettivo di produrre software privo di errori. Questo risultato si dovrebbe ottenere attraverso l'uso di specifiche formali, l'applicazione di metodi rigorosi di verifica statica, cioè basata sull'analisi del codice e non sulla sua esecuzione, e la certificazione per mezzo di collaudo statistico indipendente. Il software viene prodotto come una serie di incrementi, ciascuno dei quali viene specificato, realizzato e collaudato ripetutamente finché non supera le prove di certificazione, dopo di che l'incremento può essere consegnato all'utente che lo convalida, eventualmente chiedendo una modifica delle specifiche e ripetendo il ciclo. Quando un incremento è accettato, si passa all'incremento successivo.

Più in dettaglio:

- Le attività di specifica, sviluppo e certificazione (verifica) sono affidate a tre gruppi distinti.
- Il software viene sviluppato in cicli successivi, ognuno dei quali produce un *incremento* del prodotto.
- In ogni ciclo vengono definiti i requisiti dell'incremento, e in base a questi requisiti il gruppo di sviluppo produce il codice ed il gruppo di certificazione prepara i dati di prova (*test cases*). Il gruppo di sviluppo *non esegue i programmi* e si affida unicamente a metodi di analisi statica per valutare la correttezza del codice. Quando il gruppo di sviluppo confida di aver prodotto del codice corretto, lo consegna al gruppo di certificazione che prova il sistema complessivo (insieme di incrementi ottenuti fino a quel momento), certificandone l'affidabilità statisticamente e restituendo i risultati al gruppo di sviluppo. Questo modifica il software e fornisce al gruppo di certificazione una *notifica di cambiamento del progetto* (*engineering change notice*). Il ciclo viene quindi ripetuto.
- Il processo termina quando si raggiunge un'affidabilità accettabile.

L'affidabilità viene quantificata misurando la frequenza dei guasti nel corso del collaudo. Il collaudo è una simulazione dell'uso del software basata sul *profilo operativo* dell'applicazione, cioè uno schema tipico delle interazioni degli utenti col sistema. Per esempio, se si deve valutare l'affidabilità di un sito web si può ricavare un profilo operativo dalla storia degli accessi a siti web simili: questo profilo dice che tipo di richieste

vengono fatte e con quale frequenza, e con queste informazioni si può simulare il carico del nuovo sito.

Appendice B

Formalismi di specifica

B.1 Modello Entità–Relazioni

Il modello Entità–Relazioni (ER) è un modello descrittivo semiformale per applicazioni orientate ai dati, di tipo semantico. Permette di descrivere la struttura concettuale dei dati, cioè le relazioni logiche fra gli oggetti del "mondo reale" rappresentati dai dati, indipendentemente sia dalle operazioni che devono essere eseguite sui dati (p.es., ricerca, ordinamento,...), che dalla loro implementazione.

Un'*entità* rappresenta un insieme di oggetti, ciascuno dei quali possiede *attributi*, che rappresentano informazioni significative dal punto di vista dell'applicazione per caratterizzare ciascun oggetto. Alcuni attributi, detti *attributi chiave*, identificano ciascun elemento dell'insieme. Per esempio, l'insieme degli impiegati di una ditta si può rappresentare con l'entità *Impiegati*, definita dagli attributi *Matricola*, *Cognome*, *Nome*, e *Stipendio*. La *Matricola* è un attributo chiave, perché permette di identificare ciascun impiegato (anche in casi di omonimia).

Un'entità (fig. B.1) viene rappresentata da un rettangolo, ed i suoi attributi vengono rappresentati da ellissi, collegate al rettangolo.

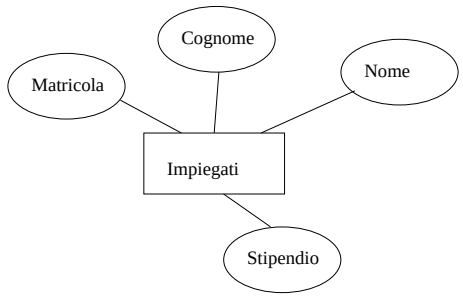


Figura B.1: Un'entità

Gli oggetti (e quindi le entità a cui appartengono) sono legati reciprocamente da

relazioni. Una relazione può sussistere fra elementi di una stessa entità (per esempio un impiegato *collabora* con altri impiegati), o fra elementi di due o piú entità.

Una relazione (fig. B.2) si rappresenta con una losanga collegata alle entità che vi partecipano. Le linee di collegamento sono decorate con frecce per rappresentare relazioni biunivoche (frecce verso due entità) o funzionali (frecce verso una sola entità). La figu-

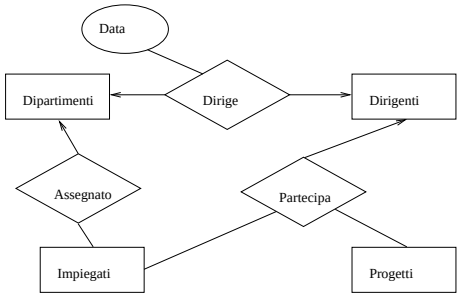


Figura B.2: Un diagramma Entità/Relazioni

ra B.2 mostra tre relazioni: ogni impiegato è assegnato a un dipartimento e partecipa ad uno o piú progetti, ogni progetto vede la partecipazione di uno o piú impiegati e di un dirigente, e infine ogni dirigente è a capo di un dipartimento. La figura mostra anche una relazione caratterizzata da un attributo, cioè la data d'inizio dell'incarico di un dirigente alla guida di un dipartimento. La relazione fra dirigenti e dipartimenti è uno a uno.

B.2 Espressioni regolari

Un linguaggio basato sulle *espressioni regolari* permette di descrivere formalmente la sintassi dei dati elaborati da un'applicazione, e per questo le espressioni regolari verranno trattate in questa sezione dedicate ai linguaggi orientati ai dati. È bene precisare che le espressioni regolari possono specificare anche sequenze di eventi o di azioni, per cui potrebbero essere trattate anche nell'àmbito dei formalismi di tipo operazionale. Osserviamo, in particolare, che esiste una stretta relazione fra le espressioni regolari e gli automi a stati finiti (sez. 4.4.1).

Un'espressione regolare descrive una famiglia di sequenze (o *stringhe*) di simboli appartenenti ad un alfabeto: piú precisamente, per *alfabeto* si intende un insieme finito di simboli elementari (che in pratica può coincidere con l'insieme di caratteri alfanumerici usati in un sistema di elaborazione, ma anche con altri tipi di informazione) comprendente l'elemento nullo (λ), e un'espressione regolare rappresenta un'insieme di stringhe per mezzo di operazioni applicate ad altri insiemi di stringhe.

Le espressioni regolari elementari su un alfabeto sono le stringhe formate da un unico simbolo. Per esempio, se l'alfabeto è l'insieme $\{a, b, c\}$, le espressioni regolari fondamentali sono **a**, **b** e **c**, che rappresentano rispettivamente i sottoinsiemi $\{a\}$, $\{b\}$ e $\{c\}$, ovvero le stringhe a, b e c.

L'operazione di *concatenazione* di due espressioni E_1 ed E_2, rappresentata di solito con la semplice giustapposizione delle espressioni, produce l'insieme delle stringhe formate da un prefisso uguale ad una stringa appartenente a E_1 seguito da un suffisso uguale ad una stringa appartenente a E_2. Per esempio, $\mathsf{ab} = \{ab\}$, ovvero l'insieme contenente solo la stringa ab.

L'operazione di *scelta* fra due espressioni E_1 ed E_2, rappresentata di solito col simbolo '|', produce l'unione degli insiemi rappresentati dalle due espressioni. Per esempio, $\mathsf{a} \mid \mathsf{b} = \{a, b\}$, $\mathsf{a}(\mathsf{b} \mid \mathsf{c}) = \{ab, ac\}$.

L'operazione di *chiusura* (anche *chiusura* o *stella di Kleene*), rappresentata col simbolo '$*$', di un'espressione E produce l'insieme formato dalla stringa nulla e dalle stringhe che si possono ottenere concatenando un numero finito (ma non limitato) di stringhe appartenenti a E. Per esempio, $\mathsf{a}* = \lambda \mid \mathsf{a} \mid \mathsf{aa} \mid \ldots = \{\lambda, a, aa, \ldots\}$.

Un'applicazione tipica di questo formalismo è la specifica degli elementi lessicali (*token*) di un linguaggio di programmazione (costanti, identificatori, parole chiave...). Tuttavia, come già accennato, i simboli dell'alfabeto usato nelle espressioni regolari possono rappresentare altri concetti, come, per esempio, azioni o eventi. Le espressioni regolari sono quindi uno strumento di specifica molto flessibile. La natura formale delle espressioni regolari fa sí che un'espressione si possa trasformare in altre espressioni equivalenti, di verificare l'equivalenza di due espressioni, e soprattutto di verificare se una particolare stringa di simboli appartiene all'insieme definito da un'espressione.

Ricordiamo inoltre che le espressioni regolari, oltre ad essere usate come linguaggio di specifica, hanno un ruolo importante nei linguaggi di programmazione (come, per esempio, il Perl) o negli strumenti (per esempio, i programmi `sed` e `grep` nei sistemi Unix) dedicati all'elaborazione di testi.

Esistono varie notazioni per le espressioni regolari, e qui descriveremo un sottoinsieme della notazione impiegata dal programma Lex [27]. Questo programma legge un file contenente la specifica di alcune espressioni regolari, a ciascuna delle quali sono associate delle elaborazioni (*azioni*), e produce un programma in C che, leggendo un file di testo, riconosce le stringhe descritte dalle espressioni regolari e, per ogni riconoscimento, esegue le azioni associate. Anche il linguaggio in cui sono scritte le espressioni regolari si chiama Lex.

In Lex, un'espressione regolare è formata da *caratteri testuali* e da *operatori*. I caratteri testuali sono lettere, cifre, simboli aritmetici e segni d'interpunzione. Gli operatori usati nelle espressioni regolari sono i seguenti:

[...] classi di caratteri, per esempio `[a-zA-Z0-9]` rappresenta l'insieme costituito dai caratteri alfabetici minuscoli e maiuscoli e dalle cifre decimali;

`.` qualsiasi carattere escluso il carattere di nuova linea;

`|` espressioni alternative (scelta), per esempio `ab|cd` rappresenta `ab` e `cd`;

`*` zero o piú volte l'espressione precedente (chiusura di Kleene);

`?` espressione opzionale, per esempio `ab?c` rappresenta `ac` e `abc`;

`+` una o piú volte l'espressione precedente;

`{ m, n }` ripetizione fra m ed n volte: `a{2}` riconosce `aa`, `a{2,4}` rappresenta `aa`, `aaa`, `aaaa`.

La concatenazione di due espressioni si esprime semplicemente scrivendole una di séguito all'altra. Osserviamo che tutti gli operatori del Lex si possono esprimere in termini degli operatori fondamentali di concatenazione, scelta e chiusura, eventualmente introducendo la stringa nulla.

A questa notazione si aggiunge la possibilità di dare un nome a delle espressioni, e di riferirsi ad esse scrivendone il nome fra parentesi graffe.

B.2.1 Esempio

Il seguente esempio è una semplice grammatica per riconoscere i token usati in espressioni di assegnamento formate da identificatori, costanti intere ed operatori aritmetici e di assegnamento. Un identificatore è formato da uno o piú caratteri, il primo dei quali è una lettera o una sottolineatura '_' e gli altri possono essere lettere, cifre o sottolineature. Un intero è costituito da una o piú cifre.

```
[a-zA-Z_][a-zA-Z_0-9]*    return( IDENTIFIER );
[0-9]+                    return( INTEGER );
"="                       return( '=' );
"+"                       return( '+' );
"-"                       return( '-' );
"*"                       return( '*' );
"/"                       return( '/' );
```

Le azioni associate al riconoscimento di stringhe appartenenti alle diverse espressioni regolari sono, in questo caso, delle semplici istruzioni `return`. Il programma Lex, partendo da questa grammatica, produrrà una funzione di riconoscimento che, leggendo un file, eseguirà l'istruzione `return` appropriata ogni volta che riconosce un token. Questa funzione potrà essere usata da un programma piú complesso, per esempio da un compilatore.

B.3 Grammatiche non contestuali

Le *grammatiche non contestuali* (*context-free grammars*) vengono usate comunemente per specificare le sintassi dei linguaggi di programmazione, che sono troppo complesse per essere descritte dal linguaggio delle espressioni regolari. Ricordiamo che una grammatica non contestuale (o *di tipo 2*) è definita da un insieme di *simboli non terminali*, che rappresentano strutture complesse (per esempio, programmi, istruzioni, espressioni), un insieme di *simboli terminali*, che rappresentano strutture elementari (per esempio, identificatori, costanti, parole chiave), e un insieme di *produzioni* della forma

$$A \to \alpha$$

dove A è un simbolo non terminale e α è una sequenza di simboli, ciascuno dei quali può essere terminale o non terminale. Una produzione di questa forma dice che il simbolo A può essere sostituito dalla sequenza α[1].

Tipicamente tali grammatiche vengono descritte con una notazione basata sulla *forma normale di Backus-Naur* (*BNF*), ben nota a chiunque abbia studiato un linguaggio di programmazione. È utile tener presente che i linguaggi che possono essere specificati non si limitano ai linguaggi di programmazione, ma comprendono anche i linguaggi di comando dei sistemi operativi e delle applicazioni, interattive, i formati dei file di configurazione, e in generale i formati di qualsiasi dato letto o prodotto da un'applicazione.

Per esempio, la seguente specifica in linguaggio YACC [27] descrive una sintassi per le espressioni aritmetiche:

```
espr : espr '+' espr
     | espr '-' espr
     | espr '*' espr
     | espr '/' espr
     | IDENTIFIER
     | INTEGER
     ;
```

Il programma Yacc, analogamente al programma Lex già visto, partendo da questa sintassi può produrre una funzione in linguaggio C che riconosce le espressioni aritmetiche in un file di testo.

B.4 ASN.1

La *Abstract Syntax Notation 1* è una notazione destinata a descrivere i dati scambiati su reti di comunicazioni. La ASN.1 è definita dallo standard ISO/IEC 8824-1:2002, derivato dallo standard CCITT X.409 (1984)[2].

Per mezzo della ASN.1 si descrive una *sintassi astratta*, cioè generica e indipendente dalle diverse implementazioni delle applicazioni che si scambiano dati specificati in ASN.1. Questo permette la comunicazione fra applicazioni eseguite su architetture diverse. Una sintassi astratta è costituita da un insieme di definizioni di tipi e definizioni di valori. Queste ultime sono definizioni di costanti. I tipi sono *semplici* e *strutturati*.

[1] Osserviamo che la sostituzione può essere applicata qualunque sia il *contesto* di A, cioè i simboli che precedono e seguono A; questo spiega il termine "non contestuali".

[2] Si veda il sito `asn1.elibel.tm.fr`.

B.4.1 Tipi semplici

I tipi semplici sono `INTEGER`, `REAL`, `BOOLEAN`, `CHARACTER STRING`, `BIT STRING`, `OCTET STRING`, `NULL`, e `OBJECT IDENTIFIER`. Per alcuni di questi tipi si possono definire dei sottotipi esprimendo delle restrizioni di vario tipo, come nei seguenti esempi:

```
Valore ::= INTEGER
Tombola ::= INTEGER (1..90)
Positive ::= INTEGER (0< ..MAX)
Negative ::= INTEGER (MIN .. <0)
Zero ::= INTEGER (0)
NonZero :: INTEGER (INCLUDES Negative | INCLUDES Positive)
```

Il tipo `Valore` coincide con `INTEGER`, i tipi `Tombola`, `Positive` e `Negative` sono sottoinsiemi di `INTEGER`. Il tipo `Zero` viene definito per enumerazione (contiene solo l'elemento `0`), ed il tipo `NonZero` viene definito come unione di altri tipi.

Di séguito diamo alcuni esempi di definizioni di tipi o valori che usano tipi semplici.

I valori del tipo `REAL` sono terne formate da mantissa, base, esponente:

```
avogadro REAL ::= {60221367, 10, 16}
```

Quest'esempio è una definizione di un valore, che specifica il tipo e il valore della costante `avogadro`.

I `BIT STRING` rappresentano stringhe di bit:

```
BinMask BIT STRING ::= '01110011'B
HexMask BIT STRING ::= '3B'H
```

I valori `BinMask` e `HexMask` vengono espressi in notazione rispettivamente binaria ed esadecimale.

Gli `OCTET STRING` rappresentano stringhe di byte, di cui si può indicare la lunghezza:

```
Message ::= OCTET STRING (SIZE(0..255))
```

Ci sono alcuni tipi predefiniti di stringhe di caratteri, come `NumericString` (cifre decimali e spazio), `PrintableString` (lettere, cifre e caratteri speciali), `GraphicString` (simboli), etc.:

```
myPhoneNumber NumericString ::= "883455"
grazieGreco GraphicString ::= "ευχαριστω"
```

Il tipo `OBJECT IDENTIFIER` serve ad assegnare nomi unici alle varie entità definite da specifiche in ASN.1, basandosi su uno schema gerarchico di classificazione chiamato *Object Identifier Tree*:

```
AttributeType ::= OBJECT IDENTIFIER
telephoneNumber AttributeType ::= {2 5 4 20}
```

In questo esempio, `telephoneNumber` è un identificatore di oggetti il cui valore è `{2 5 4 20}`, dove 2 rappresenta il dominio degli identificatori definiti congiuntamente dall'ISO e dal CCITT, 5 rappresenta il dominio degli indirizzari (*directories*), 4 rappresenta il dominio degli attributi (di una voce in un indirizzario), e 20 rappresenta i numeri telefonici. Se un'organizzazione volesse mantenere un database contenente i nomi e i numeri di telefono dei propri dipendenti, potrebbe pubblicare lo schema logico del database etichettando il tipo `telephoneNumber` con l'identificatore (*OID*) `{2 5 4 20}`. In questo modo chiunque sviluppi del software che debba interagire col database ha un riferimento univoco alla definizione del tipo `telephoneNumber`: l'identificatore permette quindi di assegnare un significato standard agli elementi di un sistema informativo.

B.4.2 Tipi strutturati

I tipi strutturati sono `ENUMERATED`, `SEQUENCE`, `SET`, e `CHOICE`.

I tipi `ENUMERATED` sono simili ai tipi enumerati del Pascal o del C++. È possibile assegnare un valore numerico alle costanti dell'enumerazione:

```
DaysOfTheWeek ::= ENUMERATED
{
    sunday(0), monday(1), ... saturday(6)
}
```

I tipi `SEQUENCE` sono simili ai record del Pascal o alle `struct` del C++. L'ordine dei campi nella sequenza è significativo.

```
WeatherReport ::= SEQUENCE
{
    stationNumber   INTEGER (1..99999),
    timeOfReport    UTCTime,
    pressure        INTEGER (850..1100),
    temperature     INTEGER (-100..60),
}
```

Il tipo `UTCTime` che appare in questo esempio è un tipo predefinito (uno *useful type*), i cui valori possono avere la forma YYMMDDHHMMSSZ oppure YYMMDDHHMMSS±HHMM, dove i secondi sono opzionali. Nella prima forma, 'Z' è, letteralmente, la lettera zeta.

Nei tipi `SET`, i valori dei campi si possono presentare in qualsiasi ordine:

```
TypeA ::= SET
{
    p BOOLEAN,
    q INTEGER,
```

```
    r BIT STRING
}

valA TypeA ::=
{
    p TRUE
    r '83F'H
    q -7
}

TypeB ::= SET
{
    r [0] INTEGER,
    s [1] INTEGER,
    t [2] INTEGER
}
```

I numeri fra parentesi quadre si chiamano *tag* e servono a distinguere componenti dello stesso tipo.

I tipi `CHOICE` sono simili alle `union` del C++:

```
typeC ::= CHOICE
{
    x [0] REAL,
    y [1] INTEGER,
    z [2] NumericString
}
```

B.4.3 Moduli

Le definizioni di una specifica in ASN.1 possono essere raggruppate in moduli. Ogni modulo è identificato da un `OBJECT IDENTIFIER` e dichiara quali tipi e valori vengono esportati o importati.

```
WeatherReporting {2 6 6 247 7} DEFINITIONS ::=
BEGIN
    EXPORTS WeatherReport;
    IMPORTS IdentifyingString FROM StationsModule { ... };
    WeatherReport ::= SEQUENCE
    {
        idString IdentifyingString,
        stationNumber (0..99999),
        ...
    }
END
```

In questo esempio si suppone che `StationsModule` sia un modulo definito altrove, identificato dal nome e dall'`OBJECT IDENTIFIER`, qui non indicato.

B.4.4 Sintassi di trasferimento

La *sintassi di trasferimento* specifica il modo in cui i dati descritti dalla sintassi astratta vengono codificati per essere trasmessi. Una sintassi di trasferimento standardizzata (ISO 8825:1987) è costituita dalle *Basic Encoding Rules* (*BER*).

B.5 Formalismi orientati alle funzioni

Le notazioni orientate alle funzioni servono a descrivere l'aspetto funzionale dei sistemi, cioè le elaborazioni che vengono compiute sui dati. In queste notazioni il sistema viene descritto in termini di blocchi funzionali collegati da flussi di dati da elaborare, e questi blocchi a loro volta vengono scomposti in blocchi piú semplici. Osserviamo che questo metodo, oltre che per la specifica dei requisiti, si presta alla specifica del progetto, cioè a descrivere come viene realizzato il sistema. In questo caso, i blocchi funzionali vengono identificati con i sottosistemi che implementano le funzioni specificate.

In questi formalismi la rappresentazione grafica si basa sui *diagrammi di flusso dei dati* (DFD), nei quali il sistema da specificare viene visto come una rete di trasformazioni applicate ai dati che vi fluiscono. I DFD permettono il *raffinamento* della specifica mediante *scomposizione*: ogni nodo (*funzione*) può essere scomposto in una sottorete. Le metodologie basate sulla scomposizione di funzioni di trasformazione dei dati sono indicate col termine *Structured Analysis/Structured Design* (SA/SD). La scomposizione di ogni funzione deve rispettare il vincolo della *continuità del flusso informativo*, cioè i flussi attraversanti la frontiera della sottorete ottenuta dall'espansione di un nodo devono essere gli stessi che interessano il nodo originale.

Ogni funzione può essere scomposta in funzioni componenti. Quando non si ritiene utile scomporre ulteriormente una funzione, questa può essere specificata in linguaggio naturale, in pseudocodice[3] o in linguaggio naturale strutturato, o in un linguaggio algoritmico simile ai linguaggi di programmazione, ad alto livello e tipato.

Gli elementi della notazione sono:

agenti esterni: rappresentati da rettangoli, sono i produttori e i consumatori dei flussi di dati all'ingresso e all'uscita del sistema complessivo.
funzioni: (o *processi*) rappresentate da cerchi, sono le trasformazioni operate sui dati.
flussi: rappresentati da frecce, sono i dati scambiati fra le funzioni, nel verso indicato.
depositi: rappresentati da doppie linee, sono memorie permanenti.

Ogni elemento viene etichettato con un nome.

La fig. B.3 mostra il primo livello della specifica di un sistema di monitoraggio dei malati in un ospedale. Una prima espansione viene mostrata in fig. B.4, e in fig. B.5 mostriamo l'espansione della bolla `monitoraggio centrale` di fig. B.4.

[3]Un linguaggio simile a un linguaggio di programmazione, spesso non specificato formalmente.

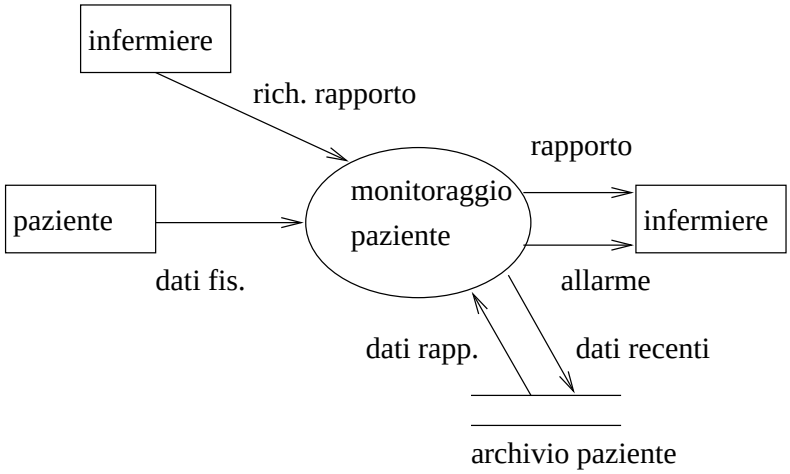


Figura B.3: Primo livello

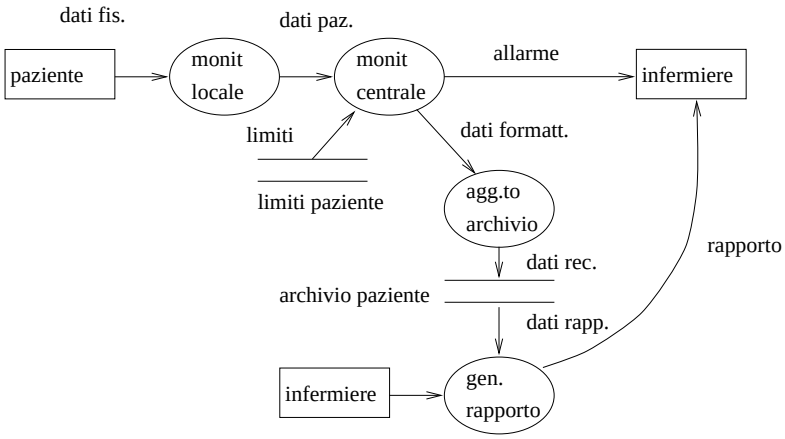


Figura B.4: Secondo livello

B.6 La notazione Z

La notazione Z [46] è uno dei piú noti formalismi di specifica basati sulla logica. Le basi di questo formalismo sono:

- una logica del primo ordine, con teoria dell'uguaglianza;
- la teoria degli insiemi (di Zermelo-Fraenkel);
- un sistema deduttivo basato sulla *deduzione naturale*;
- un *linguaggio di schemi* che permette di strutturare una specifica in un insieme di definizioni raggruppate in moduli.

La sintassi della logica usata da Z è leggermente diversa da quella usata in queste dispense. La teoria degli insiemi comprende i concetti relativi alle relazioni e alle funzioni, nonché all'aritmetica, ed usa un grande numero di operatori che permettono di esprimere sinteticamente le proprietà delle entità specificate.

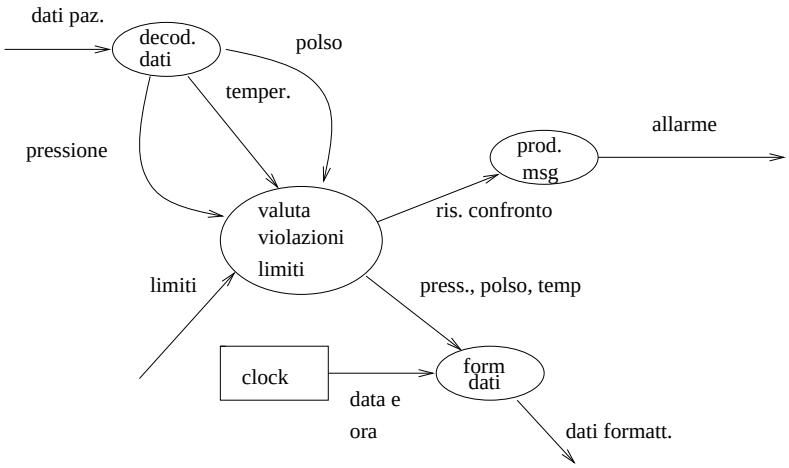


Figura B.5: Terzo livello (parziale)

Il sistema di deduzione naturale, a differenza dei sistemi minimi visti finora, sfrutta molte regole di inferenza associate agli operatori logici e matematici del linguaggio. Questo permette di costruire dimostrazioni piú facilmente.

La notazione Z comprende varie convenzioni per facilitare la specifica dei sistemi di calcolo. In particolare, lo stato di un sistema viene descritto da uno schema (gruppo di definizioni e formule logiche), e le operazioni che lo modificano vengono descritte da schemi che definiscono le variazioni dello stato in séguito alle operazioni. Opportune notazioni permettono di esprimere sinteticamente queste variazioni.

La metodologia di sviluppo basata sulla notazione Z prevede che un sistema venga descritto in termini matematici (usando concetti quali *insieme*, *sequenza*, *funzione*...), e che la specifica iniziale venga trasformata progressivamente fino ad ottenere una descrizione abbastanza dettagliata dell'implementazione. Il sistema deduttivo permette di verificare determinate relazioni o proprietà, oltre alla consistenza fra le diverse versioni della specifica ottenute nei vari passi del processo di sviluppo.

B.7 La Real-Time Logic

La Real-Time Logic (RTL) [24] è una logica tipata del primo ordine orientata all'analisi di proprietà temporali dei sistemi in tempo reale.

La semantica della RTL è un modello che riassume i concetti fondamentali dei sistemi in tempo reale:

azioni
predicati di stato
eventi
vincoli temporali

Le azioni possono essere *semplici* o *composte*, possono essere eseguite in sequenza o in parallelo, e possono essere soggette a vincoli di sincronizzazione.

I predicati di stato sono asserzioni sullo stato del sistema fisico di cui fa parte il software da analizzare.

Gli eventi sono suddivisi in *eventi esterni* (causati dall'ambiente esterno), *eventi iniziali* che segnano l'inizio di un'azione, *eventi finali* che ne segnano la fine, *eventi di transizione* che segnano i cambiamenti in qualche attributo del sistema.

I vincoli temporali sono asserzioni sui valori assoluti degli istanti in cui si verificano gli eventi. I vincoli possono essere *periodici* se prescrivono che le occorrenze di un evento si verifichino con una certa frequenza, *sporadici* se prescrivono che un'occorrenza di un evento avvenga entro un certo tempo dopo l'occorrenza di un altro evento.

La sintassi dell'RTL comprende delle costanti che denotano eventi, azioni, e numeri interi, ed operatori per rappresentare la struttura delle azioni composte. La *funzione di occorrenza* $@(e, i)$ denota l'istante dell'i-esima occorrenza dell'evento e. I predicati di stato vengono espressi specificando l'intervallo di tempo in cui sono (o devono essere) veri. Il linguaggio dell'RTL include i comuni connettivi logici e gli operatori aritmetici e relazionali.

La parte deduttiva della RTL si basa su una serie di assiomi che descrivono le proprietà fondamentali di eventi e azioni, ricorrendo alla funzione di occorrenza. Per esempio, la funzione di occorrenza stessa soddisfa, per ogni evento E, i seguenti assiomi:

$$\forall i @(E, i) = t \Rightarrow t \geq 0$$
$$\forall i \forall j (@(E, i) = t \wedge @(E, j) = t' \wedge i < j) \Rightarrow t < t'$$

Questi assiomi formalizzano due semplici nozioni: ogni occorrenza di un evento avviene in un istante successivo all'istante $t = 0$ preso come riferimento, e due occorrenze successive di uno stesso evento avvengono in istanti successivi.

Un sistema viene specificato attraverso la sua descrizione in RTL: questa descrizione, insieme agli assiomi, costituisce una teoria rispetto alla quale si possono verificare delle proprietà del sistema, in particolare le *asserzioni di sicurezza* che rappresentano i vincoli temporali che devono essere rispettati dal sistema. Per esempio, la seguente formula specifica che un'operazione di lettura ($READ$) deve durare meno di 10 unità di tempo:

$$\forall i @(\uparrow READ, i) + 10 \geq @(\downarrow READ, i)$$

Nella formula precedente, i simboli $\uparrow READ$ e $\downarrow READ$ rappresentano, rispettivamente, l'evento iniziale e l'evento finale dell'azione $READ$. Un altro esempio riguarda la risposta A del sistema ad ogni occorrenza dell'evento sporadico E, nell'ipotesi che le occorrenze successive di E siano separate da un intervallo minimo s e che A debba terminare entro la scadenza (*deadline*) d:

$$\forall i @(E, i) + s \leq @(E, i + 1)$$
$$\forall i \exists j @(E, i) \leq @(\uparrow A, j) \wedge @(\downarrow A, j) \leq @(E, i) + d$$

B.8 Reti di Petri

Le reti di Petri sono un formalismo operazionale per la descrizione di sistemi, le cui caratteristiche di astrattezza e generalità ne permettono l'applicazione a numerosi problemi, quali, per esempio, la valutazione delle prestazioni, i protocolli di comunicazione, i sistemi di controllo, le architetture multiprocessor e data flow, e i sistemi fault-tolerant.

Il modello delle reti di Petri è basato sui concetti di *condizioni* e *azioni*, piuttosto che di stati e transizioni. Una rete di Petri descrive un sistema specificando l'insieme delle azioni che possono essere compiute dal sistema o dall'ambiente in cui si trova, le condizioni che rendono possibile ciascuna azione, e le condizioni che diventano vere in séguito all'esecuzione di ciascuna azione. Lo stato del sistema è definito dall'insieme delle condizioni che in un dato istante sono vere, e le transizioni da uno stato all'altro sono causate dall'esecuzione delle azioni (che anche nel vocabolario delle reti di Petri si chiamano *transizioni*).

Una rete di Petri è un grafo orientato bipartito, i cui nodi sono divisi in *posti* (*places*) e *transizioni*, e i cui archi uniscono posti a transizioni o transizioni a posti. Se un arco va da un place a una transizione il place è detto di *ingresso* alla transizione, altrimenti è di *uscita*. Ad ogni place p è associato un numero intero $M(p)$, e si dice che p è *marcato* con $M(p)$ token, o che *contiene* $M(p)$ token. La funzione M che assegna un numero di token a ciascun place è la *marcatura* della rete; questa funzione serve a modellare le condizioni che rendono possibile lo *scatto* (cioè l'esecuzione) delle transizioni. Per esempio, un certo place potrebbe modellare la condizione "*ci sono dei messaggi da elaborare*" e la sua marcatura potrebbe modellare il numero di messaggi in attesa a un certo istante. Graficamente i place vengono rappresentati da cerchi, le transizioni da segmenti o rettangoli, gli archi da frecce, e i token contenuti in un place p da $M(p)$ "pallini" dentro al place.

Piú precisamente, una rete è una terna:

$$N = \langle P, T; F \rangle$$

dove

- gli insiemi P (posti) e T (transizioni) sono disgiunti:

$$P \cap T = \emptyset$$

- la rete non è vuota:

$$P \cup T \neq \emptyset$$

- F è una relazione che mappa posti in transizioni e transizioni in posti:

$$F \subseteq (P \times T) \cup (T \times P)$$

La *relazione di flusso* F può essere espressa per mezzo di due funzioni pre() e post() che associano, rispettivamente, gli elementi di ingresso e gli elementi di uscita a ciascun elemento della rete, sia esso un posto o una transizione. Possiamo considerare separatamente le restrizioni delle due funzioni ai due insiemi di nodi, usando per semplicità lo stesso nome della rispettiva funzione complessiva:

- Pre-insieme di un posto[4]:

$$\text{pre} : P \to 2^T$$

$$\text{pre}(p) = {}^\bullet p = \{t \in T \mid \langle t, p\rangle \in F\}$$

- Pre-insieme di una transizione:

$$\text{pre} : T \to 2^P$$

$$\text{pre}(t) = {}^\bullet t = \{p \in P \mid \langle p, t\rangle \in F\}$$

- Post-insieme di un posto:

$$\text{post} : P \to 2^T$$

$$\text{post}(p) = p^\bullet = \{t \in T \mid \langle p, t\rangle \in F\}$$

- Post-insieme di una transizione:

$$\text{post} : P \to 2^P$$

$$\text{post}(y) = y^\bullet = \{z \in P \mid \langle y, z\rangle \in F\}$$

La rete cosí definita è semplicemente la descrizione di un grafo, data elencandone i nodi e gli archi col relativo orientamento (una coppia $\langle x, y\rangle$ è un arco orientato da x a y), quindi la terna $\langle P, T; F\rangle$ rappresenta solo la struttura della rete. Per rappresentare lo stato iniziale del sistema modellato dalla rete si introduce la funzione *marcatura iniziale*, oltre a una *funzione peso* il cui significato verrà chiarito piú oltre.

Una *Rete Posti/Transizioni* è quindi una quintupla:

$$N_{P/T} = \langle P, T; F, W, M_0\rangle$$

dove

- la funzione W è il *peso* (o *molteplicità*) degli archi:

$$W : F \to \mathbb{N} - \{0\}$$

- la funzione M_0 è la *marcatura iniziale*:

$$M_0 : P \to \mathbb{N}$$

B.8.1 Abilitazione e regola di scatto

La marcatura iniziale descrive lo stato iniziale del sistema: in questo stato, alcune azioni saranno possibili e altre no. Delle azioni possibili, alcune (o tutte) verranno eseguite, in un ordine non specificato (e non specificabile in questo formalismo). L'esecuzione di queste

[4]la notazione 2^T rappresenta l'*insieme potenza* di T, cioè l'insieme dei suoi sottoinsiemi.

azioni modificherà lo stato del sistema, che verrà descritto da una nuova funzione marcatura, ovvero cambierà il valore associato a ciascun place (valore chiamato convenzionalmente "numero di token").

Come già detto, le azioni sono modellate dalle transizioni. Quando l'azione corrispondente ad una transizione è possibile nello stato attuale, si dice che la transizione è *abilitata* nella marcatura attuale; quando l'azione viene eseguita si dice che la transizione *scatta*.

Una transizione t è *abilitata nella marcatura* M se e solo se:

$$\forall_{p \in {}^\bullet t}\ M(p) \geq W(\langle p, t \rangle)$$

cioè, per ogni posto di ingresso alla transizione, si ha che la sua marcatura è maggiore o uguale al peso dell'arco che lo unisce alla transizione. Nel caso che ${}^\bullet t$ sia vuoto, la transizione è sempre abilitata.

La funzione peso è un mezzo per esprimere semplicemente certe condizioni. Supponiamo, per esempio, che un sistema operativo, per ottimizzare l'accesso al disco, raggruppi le richieste di lettura in blocchi di N operazioni: questo si potrebbe modellare con un posto la cui marcatura rappresenta il numero di richieste da servire, una transizione che rappresenta l'esecuzione delle richieste, e un arco il cui peso, uguale a N, fa sí che l'esecuzione avvenga solo quando c'è un numero sufficiente di richieste.

Graficamente, il peso di un arco si índica scrivendone il valore accanto all'arco. Se il peso non è indicato, è uguale a 1.

L'abilitazione di una transizione t in una marcatura M si índica con la formula

$$M\,[t\rangle$$

che si legge "t *è abilitata in* M".

Dopo lo scatto di una transizione la rete assume una nuova marcatura M' il cui valore è determinato dalla seguente *regola di scatto*, ove M è la marcatura precedente allo scatto:

$$\begin{array}{ll}
\forall_{p \in ({}^\bullet t - t^\bullet)} & M'(p) = M(p) - W(\langle p, t \rangle) \\
\forall_{p \in (t^\bullet - {}^\bullet t)} & M'(p) = M(p) + W(\langle t, p \rangle) \\
\forall_{p \in ({}^\bullet t \cap t^\bullet)} & M'(p) = M(p) - W(\langle p, t \rangle) + W(\langle t, p \rangle) \\
\forall_{p \in P - ({}^\bullet t \cup t^\bullet)} & M'(p) = M(p)
\end{array}$$

La regola di scatto si può cosí riassumere:

1. la marcatura di ciascun posto di ingresso viene diminuita del peso dell'arco dal posto alla transizione;

2. la marcatura di ciascun posto di uscita viene aumentata del peso dell'arco dalla transizione al posto;

3. la marcatura di ciascun posto di ingresso e di uscita varia di una quantità pari alla differenza fra i pesi dell'arco dalla transizione al posto e dell'arco dal posto alla transizione;

4. la marcatura dei posti non collegati direttamente alla transizione (quindi né d'ingresso né d'uscita) resta invariata.

Se, in una data marcatura, piú transizioni sono abilitate, ne scatta una sola, scelta (da un ipotetico esecutore della rete) in modo non deterministico.

La notazione

$$M\,[t\rangle\,M'$$

índica che la marcatura M' è il risultato dello scatto della transizione t abilitata in M.

Osserviamo che lo scatto (*firing*) di una transizione ha solo effetti locali, cambiando solo la marcatura dei posti di ingresso e di uscita della transizione stessa.

Due transizioni t_1 e t_2 possono scattare in successione se

$$M\,[t_1\rangle\,M' \wedge M'\,[t_2\rangle\,M''$$

e si dice allora che la *sequenza di scatti* t_1t_2 è abilitata in M, e si scrive

$$M\,[t_1t_2\rangle\,M''$$

In generale, rappresentiamo una sequenza di scatti di lunghezza n con la seguente notazione:

$$S^{(n)} = t_1 \ldots t_n$$

e scriviamo

$$M\,[t_1 \ldots t_n\rangle\,M^{(n)}$$

ovvero

$$M\,\left[S^{(n)}\right\rangle\,M^{(n)}$$

B.8.2 Esempio: produttore/consumatore

La figura B.6 mostra un sistema produttore/consumatore specificato per mezzo di una rete di Petri (questo esempio ed i successivi sono tratti da [16]). Il produttore ed il consumatore sono collegati da un magazzino intermedio di capacità 2. Osserviamo che, a differenza dell'esempio visto per gli automi a stati finiti, qui è possibile attribuire ciascun place ad uno dei tre sottosistemi: in questo senso la rappresentazione dello stato globale è distribuita fra i sottosistemi.

La marcatura iniziale, indicata in figura, specifica che il produttore è pronto a produrre un elemento (posto p_1), il consumatore è pronto a prelevare (posto c_1) un elemento dal magazzino (purché ci sia un elemento da prelevare), e il magazzino ha due posti liberi (posto *libero*).

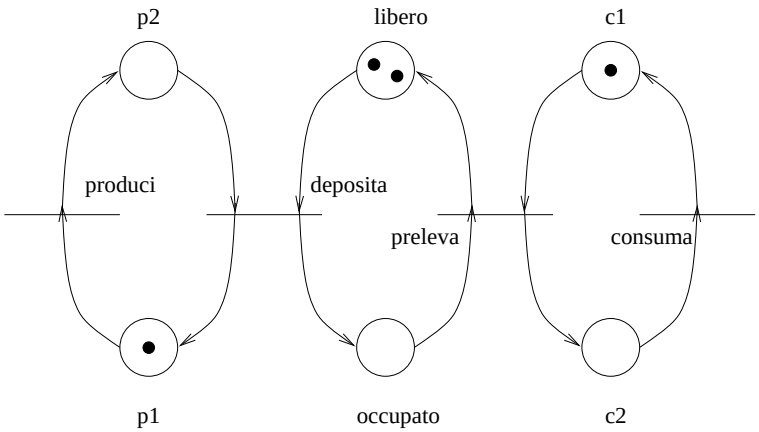


Figura B.6: pn1

B.8.3 Situazioni fondamentali

Consideriamo alcune relazioni fra transizioni, determinate sia dalla struttura della rete che dalla marcatura.

sequenza: due transizioni t ed u si dicono *in sequenza* in una marcatura M se e solo se

$$M\,[t\rangle \wedge \neg(M\,[u\rangle) \wedge M\,[tu\rangle$$

cioè u diventa abilitata solo dopo che è scattata t.

conflitto: due transizioni t ed u si dicono *in conflitto effettivo* in una marcatura M se e solo se

$$M\,[t\rangle \wedge M\,[u\rangle \wedge \exists_{p \in {}^\bullet t \cap {}^\bullet u}(M(p) < W(\langle p, t\rangle) + W(\langle p, u\rangle)) \; .$$

Questa condizione significa che tutte e due le transizioni sono abilitate, ma lo scatto di una disabilita l'altra, togliendo dei token dai place di ingresso comuni alle due transizioni.

La condizione ${}^\bullet t \cap {}^\bullet u \neq \emptyset$ si dice *conflitto strutturale*, e corrisponde all'esistenza di place di ingresso in comune fra le due transizioni.

concorrenza: due transizioni t ed u si dicono *in concorrenza effettiva* in una marcatura M se e solo se

$$M\,[t\rangle \wedge M\,[u\rangle \wedge \forall_{p \in {}^\bullet t \cap {}^\bullet u}(M(p) \geq W(\langle p, t\rangle) + W(\langle p, u\rangle)) \; .$$

Questa condizione significa che tutte e due le transizioni sono abilitate e possono scattare tutte e due in qualsíasi ordine, poiché non si influenzano a vicenda.

Un caso particolare di concorrenza effettiva è la *concorrenza strutturale*, data dalla condizione ${}^\bullet t \cap {}^\bullet u = \emptyset$; in questo caso le due transizioni non hanno place di ingresso in comune.

B.8.4 Esempio

In questo esempio modelliamo un programma concorrente costituito da tre processi. Rappresentiamo il programma in linguaggio Ada. In Ada, un processo è costituito da un

sottoprogramma di tipo *task*. Un task può contenere dei sottoprogrammi detti *entries* che possono essere invocati da altri task con delle istruzioni di *entry call*. Quando un task (cliente) invoca un'entry di un altro task (server), i due task si sincronizzano secondo il meccanismo del *rendezvous*: il corpo di una entry viene eseguito quando il server nella sua esecuzione è arrivato all'inizio dell'entry (istruzione `accept`) e un cliente è arrivato alla chiamata dell'entry. Il primo dei task che arriva al punto di sincronizzazione (rispettivamente, `accept` o entry call) aspetta che arrivi anche l'altro. Durante l'esecuzione del corpo dell'entry, il task cliente viene sospeso, e se piú clienti invocano una stessa entry, questi vengono messi in coda. Il server, con un'istruzione `select`, può accettare chiamate per piú entries: in questo caso viene eseguita la prima entry che è stata chiamata da qualche cliente.

Un'istruzione di entry call è formata dal nome del task contenente la entry, seguito da un punto, dal nome dell'entry e da eventuali parametri fra parentesi. Nel nostro esempio, i nomi dei task sono `A`, `B` e `C`, le entries sono `E1` ed `E2`, ed i corpi delle entries sono `R1` ed `R2`.

La fig. B.7 mostra il testo del programma, corrispondente alla rete di fig. B.8, in cui le transizioni rappresentano l'esecuzione delle istruzioni, ed i place rappresentano lo stato del programma fra una transizione e l'altra.

```
task body A is       |  task body B is        |      task body C is
    SA1;             |      SB1;              |          SC1;
    B.E1;            |      select            |          B.E2;
    SA2;             |          accept E1 do  |          SC2;
end A;               |              R1;       |      end C;
                     |          end E1;       |
                     |      or                |
                     |          accept E2 do  |
                     |              R2;       |
                     |          end E2;       |
                     |      end select;       |
                     |      SB2;              |
                     |  end B;                |
```

Figura B.7: Un programma concorrente (1).

Osservando la rete, possiamo mettere in evidenza la concorrenza fra le transizioni `SA1`, `SB1` ed `SC1`, il conflitto fra `accept E1` e `accept E2`, la sequenza fra `R1` e `end E1`, e fra `R2` e `end E2`.

B.8.5 Raggiungibilità delle marcature

L'analisi della *raggiungibilità* delle marcature di una rete permette di stabilire se il sistema modellato dalla rete può raggiungere determinati stati o no, e, se può raggiungerli, a quali condizioni. Permette quindi di verificare se il sistema soddisfa proprietà di *vitalità* e di *sicurezza*. La vitalità ci dice che il sistema è in grado di raggiungere le configurazioni

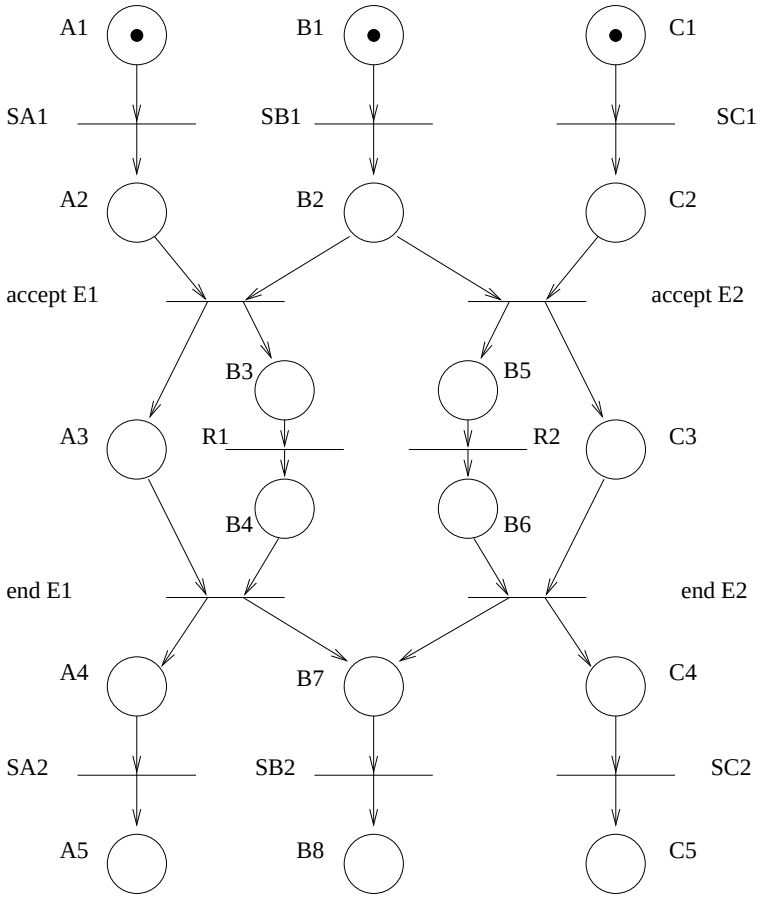


Figura B.8: Un programma concorrente (2).

desiderate: per esempio, nel sistema di controllo di un ascensore vogliamo verificare che il sistema raggiunga lo stato in cui l'ascensore e' fermo, è al livello di un piano, e le porte sono aperte. La sicurezza ci dice che il sistema non può raggiungere configurazioni indesiderate: per esempio, non vogliamo che le porte di un ascensore restino bloccate, oppure che si aprano mentre è in movimento o fra due piani.

La verifica delle proprietà di vitalità e sicurezza è possibile con qualunque linguaggio di tipo operazionale. Nelle reti di Petri, queste verifiche si basano sui concetti esposti di séguito.

Raggiungibilità in un passo. Data una rete P/T e due marcature M ed M', si dice che M' è *raggiungibile in un passo* da M se

$$\exists_{t \in T} M \, [t\rangle \, M' \; .$$

Insieme di raggiungibilità. L'*insieme di raggiungibilità* $R_N(M)$ di una rete posti/transizioni N con marcatura M è la *chiusura transitiva* della relazione di raggiungibilità in un passo della rete, cioè il piú piccolo insieme di marcature tale che:

$$M \in R_N(M)$$

$$M' \in R_N(M) \wedge \exists_{t \in T} M' [t\rangle M'' \Rightarrow M'' \in R_N(M)$$

ovvero, M appartiene all'insieme di raggiungibilità e, se M'' è raggiungibile in un passo da una marcatura M' appartenente all'insieme di raggiungibilità, allora M'' vi appartiene.

Grafo di raggiungibilità. Il *grafo di raggiungibilità* di una rete P/T è un grafo i cui nodi sono gli elementi dell'insieme di raggiungibilità, e gli archi sono le transizioni fra una marcatura all'altra. La rete viene cosí tradotta in un automa a stati, generalmente infinito.

B.8.6 Vitalità delle transizioni

Le proprietà di vitalità di un sistema si possono definire, come abbiamo visto nel paragrafo precedente, in termini di raggiungibilità delle marcature, cioè della possibilità che il sistema assuma determinate configurazioni. Un altro modo di caratterizzare la vitalità si riferisce alla possibilità che il sistema esegua determinate azioni. La *vitalità* di una transizione descrive la possibilità che la transizione venga abilitata, ed è quindi una proprietà legata alla capacità del sistema di funzionare senza bloccarsi. Si definiscono cinque gradi di vitalità per una transizione:

grado 0: la transizione non può mai scattare.
grado 1: esiste almeno una marcatura raggiungibile in cui la transizione può scattare.
grado 2: per ogni intero n esiste almeno una sequenza di scatti in cui la transizione scatta almeno n volte; è quindi possibile, per qualsíasi n, scegliere una sequenza in cui la transizione scatta n volte, ma poi la transizione può non scattare piú.
grado 3: esiste almeno una sequenza di scatti in cui la transizione scatta infinite volte.
grado 4: per ogni marcatura M raggiungibile da M_0, esiste una sequenza di scatti che a partire da M abilita la transizione (si dice che la transizione è *viva*).

Osserviamo che per una transizione vitale al grado 3 esiste almeno una sequenza si scatti (e quindi di marcature) in cui scatta infinite volte, però sono possibili anche sequenze di scatti in cui ciò non accade, perché portano la rete in una marcatura a partire da cui la transizione non può piú scattare. Per una transizione viva (grado 4) è sempre possibile portare la rete in una marcatura ove la transizione è abilitata.

Una rete P/T si dice *viva* se e solo se tutte le sue transizioni sono vive.

Consideriamo, per esempio, la rete di fig. B.9. Questa rete modella due processi A e B che devono usare in modo concorrente due risorse; queste ultime sono rappresentate dai place R_1 ed R_2. Dalla marcatura iniziale è possibile arrivare, per esempio attraverso la sequenza di scatti $[t_1 t_2 t'_1 t'_2]$ ad una marcatura in cui $M(p_3) = M(p'_3) = 1$ e la marcatura degli altri place è zero. In questa marcatura le transizioni t_3 e t'_3 non possono scattare e inoltre non è possibile raggiungere una marcatura in cui esse siano abilitate: in questa marcatura hanno una vitalità di grado zero. Questo significa che ciascuno dei due processi ha acquisito una delle due risorse e nessuno può procedere, non potendo acquisire la risorsa mancante. Questa è una tipica situazione di *deadlock*.

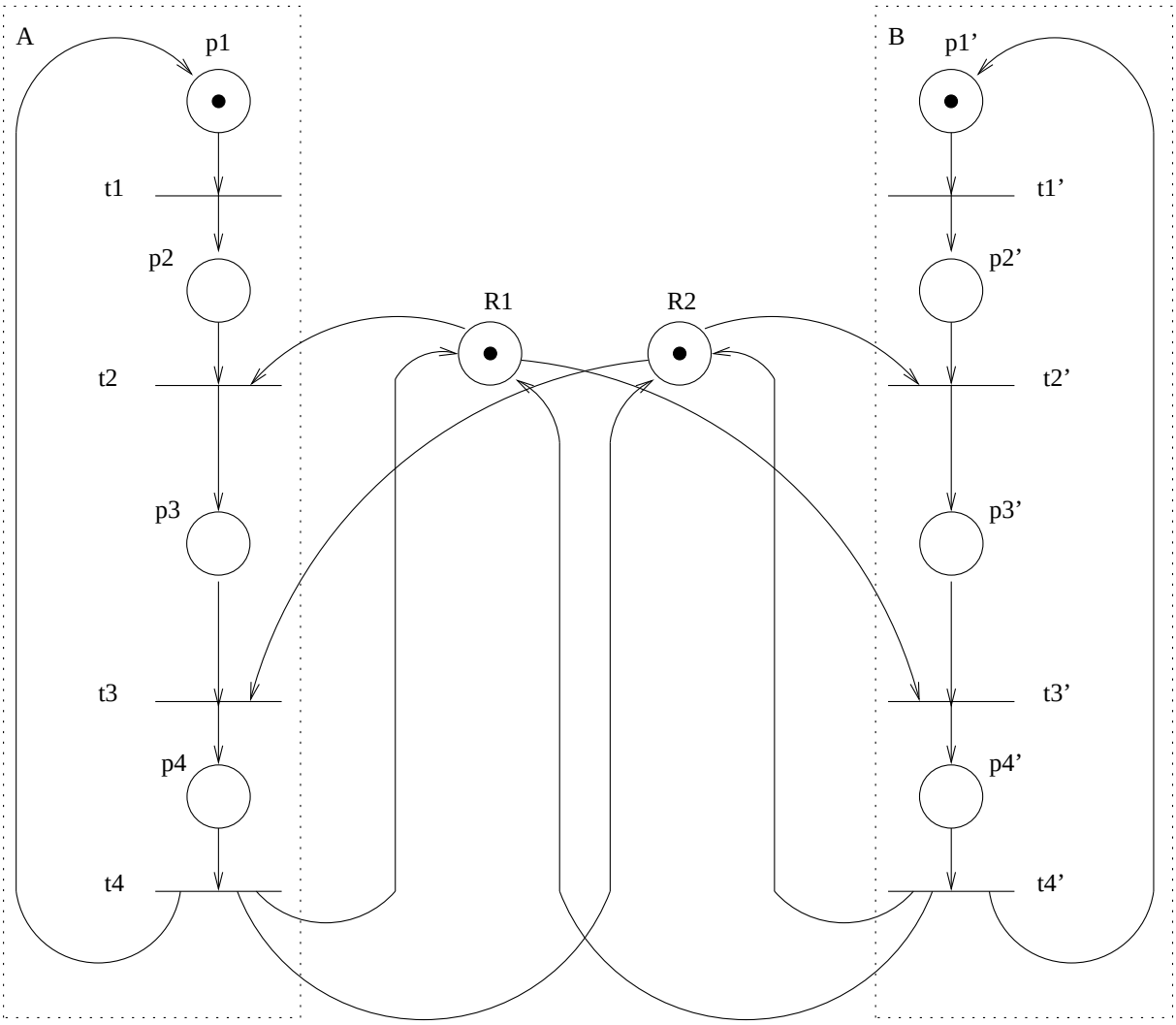


Figura B.9: Un sistema con possibilità di deadlock.

Si può evitare il deadlock modificando il sistema in modo che ciascun processo acquisisca le due risorse in un solo passo. Questo comportamento è modellato dalla rete di fig. B.10. In questo caso tutte le transizioni hanno vitalità di grado 4. Ricordiamo però che l'abilitazione di una transizione non implica il suo scatto effettivo: è possibile che il conflitto fra t_2 e t'_2 venga sempre risolto a favore della stessa transizione. Questa è una situazione di *starvation*.

B.8.7 Reti limitate

Nel modello generale delle reti di Petri non si pongono limiti al valore della marcatura di ciascun place, ma ciascuna rete, a seconda della sua struttura e della marcatura iniziale, può essere tale che esistano limiti alla marcatura di ciascun place oppure no. Molto spesso

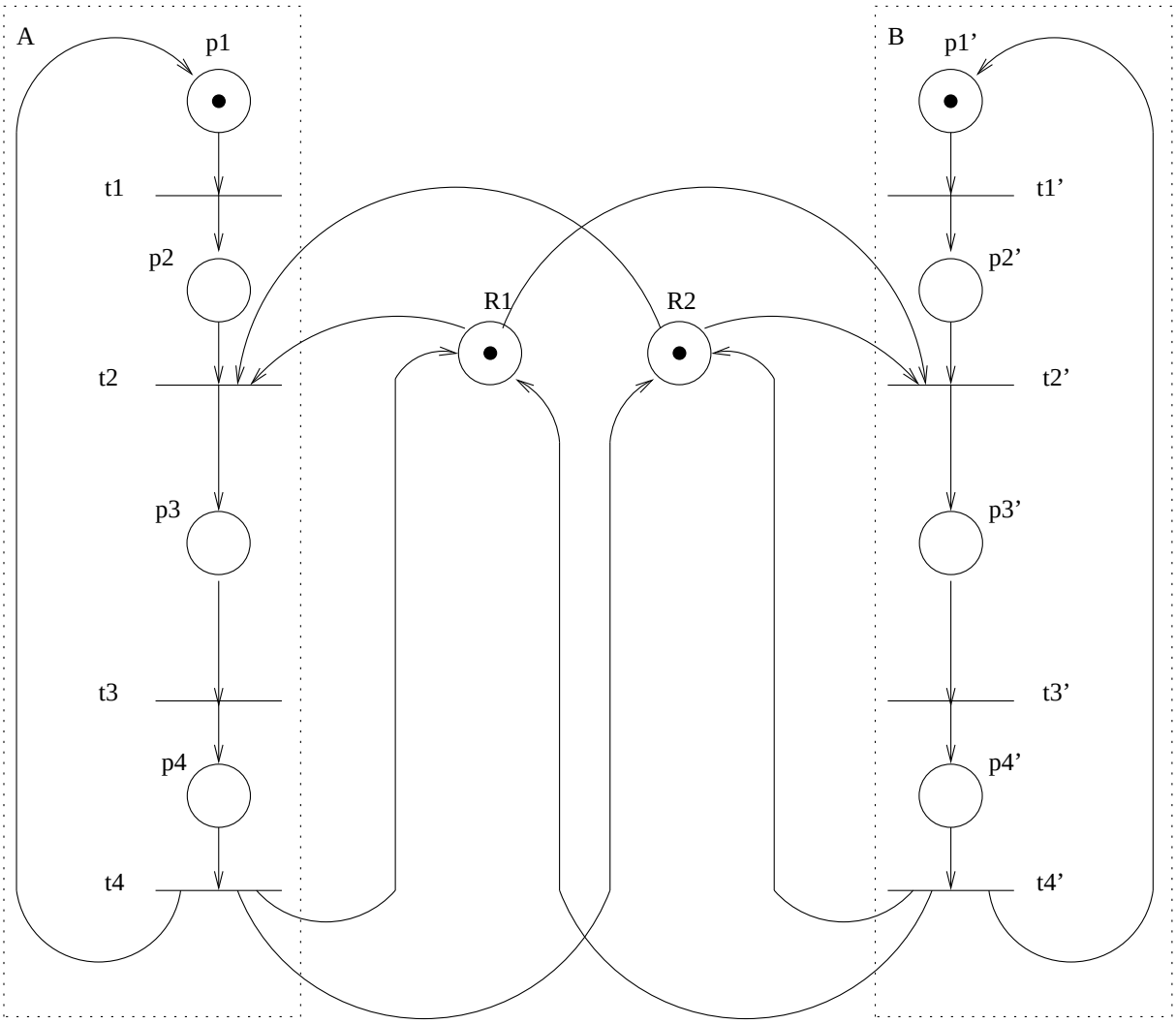


Figura B.10: Una soluzione per evitare il deadlock.

l'assenza di limiti nella marcatura di un place índica un situazione indesiderata nel sistema modellato (per esempio, il traboccamento di qualche struttura dati, o il sovraccarico di qualche risorsa), per cui è utile formalizzare il concetto di limitatezza in modo da poter verificare questa proprietà.

Un posto di una rete P/T si dice *k-limitato* (*k-bounded*) se in qualunque marcatura raggiungibile il numero di token non supera k.

Una rete P/N con marcatura M si dice *limitata* (*bounded*) se e solo se

$$\exists_{k \in \mathbb{N}} \forall_{M' \in R_N(M)} \forall_{p \in P} M'(p) \leq k$$

cioè se esiste un intero k tale che, in ogni marcatura raggiungibile da M, la marcatura di ciascun place sia non maggiore di k.

Se $k = 1$, la rete si dice *binaria*.

B.8.8 Rappresentazione matriciale

È possibile rappresentare una rete di Petri, con le sue possibili marcature e le sequenze di scatti, per mezzo di matrici e vettori. Questa rappresentazione permette di ricondurre l'analisi della rete a semplici operazioni su matrici, facilitando quindi l'uso di strumenti automatici.

Le matrici *di ingresso* I e *di uscita* O di una rete sono due matrici rettangolari di dimensione $|P| \times |T|$ ($|P|$ righe per $|T|$ colonne)[5]; l'elemento $I_{i,j}$ della matrice di ingresso è il peso dell'arco dal posto p_i alla transizione t_j, e l'elemento $O_{i,j}$ della matrice di uscita è il peso dell'arco dalla transizione t_j al posto p_i, ossia:

$$\begin{array}{ll} \forall_{\langle p_i,t_j\rangle\in F} & I_{i,j} = W(\langle p_i, t_j\rangle) \\ \forall_{\langle p_i,t_j\rangle\notin F} & I_{i,j} = 0 \\ \forall_{\langle t_j,p_i\rangle\in F} & O_{i,j} = W(\langle t_j, p_i\rangle) \\ \forall_{\langle t_j,p_i\rangle\notin F} & O_{i,j} = 0 \end{array}$$

Osserviamo che ogni colonna di queste matrici è associata ad una transizione, e che ciascun elemento della colonna rappresenta la variazione della marcatura causata dallo scatto della transizione per ogni place di ingresso o di uscita.

La marcatura M viene rappresentata da un vettore colonna m le cui componenti sono le marcature dei singoli posti:

$$m_i = M(p_i)$$

Lo scatto di una transizione t_j nella marcatura m produce la nuova marcatura[6]

$$m' = m - I_{\bullet j} + O_{\bullet j}$$

ovvero, introducendo la *matrice d'incidenza* $C = O - I$,

$$m' = m + C_{\bullet j}$$

Queste equazioni significano che, per ciascun place, la nuova marcatura è uguale alla precedente, meno i pesi degli archi uscenti dal place, piú i pesi degli archi entranti; sono cioè una riformulazione della regola di scatto.

Per ogni sequenza di scatti S si definisce un vettore colonna s di dimensione $|T| \times 1$, in cui ogni componente s_j riporta il numero di volte che la transizione t_j è scattata nella sequenza S. Quindi se $M\,[S\rangle\,M'$, vale l'*equazione fondamentale*

$$m' = m + Cs$$

[5]Ricordiamo che $|\mathcal{S}|$ è la cardinalità dell'insieme $\mathcal{S}$, cioè, per un insieme finito, il numero dei suoi elementi

[6]La notazione $A_{\bullet i}$ rappresenta la i-esima colonna della matrice A.

B.8.9 Reti conservative

Spesso ci interessa verificare che il numero di token in una rete o in una parte di essa rimanga costante. Una rete si dice quindi *conservativa* se in un suo sottoinsieme di posti la somma dei token contenuti è costante. Il sottoinsieme è individuato da una *funzione peso* che dice quali posti escludere dal conto. Questa funzione molto spesso assegna il peso uno ai posti appartenenti all'insieme e il peso zero ai posti da escludere, ma nel caso piú generale il peso può assumere valori nell'insieme dei numeri relativi, per modellare situazioni in cui la marcatura di certi place deve essere contata piú di una volta, oppure deve essere sottratta dal numero totale (ha quindi un peso negativo).

Data quindi una funzione peso H

$$H : P \to Z$$

una rete P/T con marcatura M si dice *conservativa rispetto alla funzione* H se e solo se

$$\forall_{M' \in R_N(M)} \quad \sum_{p \in P} H(p)M'(p) = \sum_{p \in P} H(p)M(p)$$

Un caso particolare è la rete *strettamente conservativa*, in cui tutti i posti hanno peso unitario, per cui si considera tutta la rete invece di un suo sottoinsieme proprio.

Se una rete è rappresentata in forma matriciale, H viene rappresentata da un vettore riga (*P-vettore*) h tale che $h_i = H(p_i)$. Un *P-invariante* è un P-vettore tale che il suo prodotto scalare per qualsiasi marcatura raggiungibile sia costante. Un P-invariante è quindi una particolare funzione H che individua una sottorete conservativa. Per trovare un tale vettore bisogna risolvere in h la seguente equazione:

$$hm' = hm$$

per qualsiasi marcatura m' raggiungibile da m, e quindi per qualsiasi sequenza di scatti. Dall'equazione fondamentale otteniamo

$$h(m + Cs) = hm$$

$$hCs = 0$$

$$hC = \mathbf{0}$$

dove $\mathbf{0}$ è un vettore nullo.

La soluzione in h di questa equazione è formata, per le reti vive, da un insieme di vettori e dalle loro combinazioni lineari.

Come applicazione di questi concetti, consideriamo due processi che devono accedere ad una risorsa in mutua esclusione. Nella rete mostrata in fig. B.11, i place p_2 e p_3 rappresentano le sezioni critiche dei due processi, le transizioni t_1 e t_4 rappresentano l'ingresso dei due processi nelle rispettive sezioni critiche, e le transizioni t_2 e t_5 ne rappresentano l'uscita. Il place p_7 rappresenta la disponibilità della risorsa.

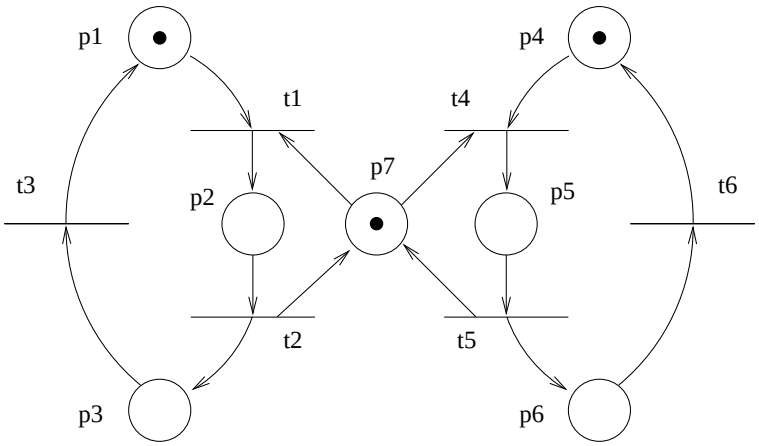


Figura B.11: pn3

Con la numerazione dei posti e delle transizioni sú esposta, la matrice di incidenza e la marcatura iniziale sono

$$C = \begin{bmatrix} -1 & 0 & 1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 \\ -1 & 1 & 0 & -1 & 1 & 0 \end{bmatrix}, \; m = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

I sottoinsiemi di posti che ci interessano sono $P_1 = \{p_1, p_2, p_3\}$, $P_2 = \{p_4, p_5, p_6\}$, e $P_3 = \{p_2, p_7, p_5\}$. Le sottoreti P_1 e P_2 rappresentano, rispettivamente, gli stati dei due processi: poiché i processi sono sequenziali, il numero di token nelle sottoreti corrispondenti è necessariamente uguale a uno, altrimenti ci sarebbe un errore nella modellazione dei processi per mezzo della rete di Petri. La sottorete P_3 permette di modellare la mutua esclusione: deve contenere un solo token, che appartiene o a p_7 (risorsa libera), oppure a uno solo dei place p_2 e p_5 (risorsa occupata da uno dei processi).

I tre sottoinsiemi sono rappresentati dai P-vettori

$$\begin{aligned} h^{(1)} &= \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \\ h^{(2)} &= \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix} \\ h^{(3)} &= \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} \end{aligned}$$

Possiamo verificare che questi vettori sono soluzioni del sistema omogeneo $hC = \mathbf{0}$, e pertanto i prodotti

$$h^{(i)}m, \quad i = 1, \ldots, 3$$

sono costanti al variare di m sulle marcature raggiungibili, ed uguali ad uno. Questo implica, fra l'altro, che viene soddisfatta la condizione di mutua esclusione.

Dai P-invarianti si possono ricavare molte informazioni. In particolare, se h è un P-invariante rispetto ad una marcatura m, e si ha che

$$hm' \neq hm$$

allora la marcatura m' non è raggiungibile da m.

B.8.10 Sequenze di scatti cicliche

Una sequenza di scatti si dice *ciclica* se riporta la rete alla marcatura iniziale. Come abbiamo visto, una sequenza di scatti si può rappresentare con un vettore (*T-vettore*) che associa a ciascuna transizione il numero di volte che essa scatta nella sequenza. Dall'equazione fondamentale

$$m' = m + Cs$$

si ottiene che i T-vettori soluzioni dell'equazione

$$Cs = \mathbf{0}$$

rappresentano sequenze di scatti cicliche. Tali vettori sono i *T-invarianti* della rete.

Nell'esempio sulla mutua esclusione, un T-invariante è

$$s = \begin{bmatrix} 3 \\ 3 \\ 3 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

Appendice C

Metriche del software

Tutte le attività industriali richiedono che le proprietà dei prodotti e dei semilavorati, oltre che delle risorse e dei processi impiegati, vengano misurate. Per esempio, nell'industria meccanica possiamo considerare le dimensioni e le proprietà tecnologiche dei pezzi (prodotti e semilavorati), le dimensioni, il consumo energetico e la velocità delle macchine operatrici (risorse materiali), il numero e lo stipendio degli addetti (risorse umane), il tempo di produzione, i tempi morti e la produttività dei cicli di lavorazione (processi). Tutte queste grandezze, o *metriche*, sono necessarie per pianificare il lavoro e valutarne i risultati: è difficile immaginare un'attività ingegneristica che possa fare a meno di metriche.

Anche da questo punto di vista, l'industria del software è abbastanza anomala, in quanto l'uso di metriche è molto meno evoluto che negli altri settori. Questo dipende in parte dalla natura "immateriale" del software, che rende difficile la stessa definizione di grandezze misurabili. Questo non vuol dire che manchino delle metriche per il software: in effetti ne sono state proposte circa 500. Purtroppo non è chiara l'utilità effettiva di certe metriche, e spesso non si trova una giustificazione teorica per le metriche effettivamente usate in pratica. L'argomento delle metriche del software è quindi complesso ed in continua evoluzione.

Per definire una metrica, bisogna innanzitutto individuare le *entità* che ci interessano, e gli *attributi* di tali entità che vogliamo quantificare. Nell'ingegneria del software le entità possono essere *prodotti* (inclusi i vari semilavorati, o *artefatti*), *risorse*, e *processi*. Alcuni prodotti sono i documenti di specifica e di progetto, il codice sorgente, i dati ed i risultati di test. Fra le risorse ricordiamo il personale e gli strumenti hardware e software, come esempi di processi citiamo le varie fasi del ciclo di vita.

Gli attributi possono essere *interni* o *esterni*. Gli attributi interni di un'entità possono essere valutati considerando l'entità isolatamente, mentre gli attributi esterni richiedono che l'entità venga osservata in relazione col proprio ambiente. Generalmente gli attributi esterni sono i piú interessanti dal punto di vista dell'utente e da quello del produttore, ma non sono misurabili direttamente e vengono valutati a partire dagli attributi interni. Per esempio, nei documenti di specifica e di progetto possiamo misurare gli attributi

interni *dimensione* e *modularità*. Questi influenzano gli attributi esterni *comprensibilità* e *manutenibilità*: si suppone che un documento breve e ben strutturato sia piú comprensibile e facile da modificare di uno lungo e caotico.

Dei vari attributi ci può interessare una *valutazione* o una *predizione* (o *stima*). La stima di una metrica si ottiene a partire dai valori di altre metriche, applicando un *modello*, cioè una formula o algoritmo.

Nel séguito tratteremo alcune metriche, scelte fra le piú diffuse o piú citate in letteratura.

C.1 Linee di codice

La metrica piú semplice e di piú diffusa applicazione è costituita dal numero di linee di codice (LOC) di un programma. Quando si usa questa metrica, bisogna specificare come vengono contate le linee di codice: per esempio, bisogna decidere se contare soltanto le istruzioni eseguibili o anche le dichiarazioni. Generalmente i commenti non vengono contati. Nei linguaggi che permettono di avere piú istruzioni su uno stesso rigo o di spezzare un'istruzione su piú righi, si può usare il numero di istruzioni invece del numero di linee di codice, ma anche il numero di istruzioni può essere difficile da definire.

Questa metrica è alla base di vari indici, quali la *produttività* (linee di codice prodotte per persona per unità di tempo), la *qualità* (numero di errori per linea di codice), il *costo unitario* (costo per linea di codice). Possiamo notare che l'uso di questi indici per valutare la qualità del software penalizza i programmi brevi.

C.2 Software science

La teoria della *Software science* propone un insieme di metriche basate su quattro parametri del codice sorgente:

η_1	numero di operatori distinti
η_2	numero di operandi distinti
N_1	numero di occorrenze di operatori
N_2	numero di occorrenze di operandi

Gli operatori comprendono, oltre agli operatori aritmetici, le parole chiave per la costruzione di istruzioni composte e le chiamate di sottoprogramma. Da questi parametri si possono ottenere metriche sia effettive che stimate.

La *lunghezza* del programma è la somma dei numeri di occorrenze di operatori ed operandi:

$$N = N_1 + N_2$$

La lunghezza stimata si ricava invece dai numeri di simboli distinti:

$$\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$$

(questa formula discende da certe ipotesi sulla struttura del programma).

Il *volume* del programma è il numero minimo di bit necessari a rappresentarlo:

$$V = N \log_2(\eta_1 + \eta_2)$$

Il *volume potenziale* del programma è il numero di bit necessari a rappresentare il programma se questo fosse ridotto ad un'unica chiamata di funzione. Questo dato si ottiene dalla formula:

$$V^* = (2 + \eta_2^*) \log_2(2 + \eta_2^*)$$

dove η_2^* rappresenta il numero di argomenti della funzione equivalente al programma.

Il *livello* del programma, un indicatore del suo grado di astrazione, è

$$L = \frac{V^*}{V}$$

Il livello può essere stimato da:

$$\hat{L} = \frac{2\eta_2}{\eta_1 N_2}$$

Lo *sforzo* (*effort*) è una misura dell'impegno psichico richiesto dalla scrittura del programma. Questa misura si basa su un modello alquanto semplicistico dei processi mentali del programmatore, e viene cosí calcolata:

$$E = \frac{V}{L} = \frac{V^2}{V^*}$$

Dallo sforzo si ricaverebbe una stima del tempo richiesto per scrivere il programma:

$$\hat{T} = \frac{E}{S}$$

dove S sarebbe il numero di decisioni elementari che un programmatore può compiere in un secondo, per cui si usa tipicamente il valore 18.

C.3 Complessità

La complessità è un concetto molto importante per l'ingegneria del software, ma è difficile da definire e quindi da misurare. Le metriche di complessità sono quindi un argomento su cui la ricerca e il dibattito scientifico sono particolarmente vivi.

C.3.1 Numero ciclomatico

Il *numero ciclomatico* è un concetto della teoria dei grafi che, applicato al grafo di controllo di un programma, dà una misura della sua complessità.

In un grafo contenente n archi, un cammino può essere rappresentato da un vettore: se gli archi sono numerati da 1 a n, un cammino viene rappresentato da un vettore V in cui V_i è il numero di volte che l'i-esimo arco compare nel cammino. Un cammino è combinazione lineare di altri cammini se il suo vettore è combinazione lineare dei vettori associati agli altri cammini. Due o piú cammini si dicono linearmente indipendenti se nessuno di essi è combinazione lineare degli altri. Nel grafo della figura C.1, i cammini

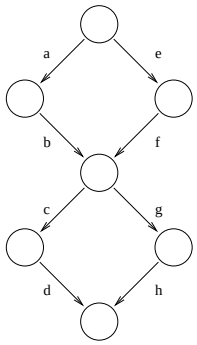


Figura C.1: Cammini linearmente indipendenti

possibili sono rappresentati dalla seguente tabella:

	a	b	c	d	e	f	g	h
(a,b,c,d)	1	1	1	1	0	0	0	0
(e,f,g,h)	0	0	0	0	1	1	1	1
(a,b,g,h)	1	1	0	0	0	0	1	1
(e,f,c,d)	0	0	1	1	1	1	0	0

Il cammino $(e, f, c, d) = (a, b, c, d) + (e, f, g, h) - (a, b, g, h)$ è una combinazione lineare degli altri tre, cioè non ne è linearmente indipendente.

Il numero ciclomatico di un programma è il numero di cammini completi (cioè cammini fra il nodo iniziale ed il nodo finale) linearmente indipendenti del grafo di controllo, modificato aggiungendo un arco fra il nodo terminale ed il nodo iniziale (in modo da rendere il grafo fortemente connesso). Se e è il numero degli archi ed n è il numero dei nodi, il numero ciclomatico è

$$v = e - n + 2$$

Se un programma è composto di p sottoprogrammi, la formula diventa

$$v = e - n + 2p$$

Il numero ciclomatico può essere calcolato anche senza costruire il grafo di controllo. Se d è il numero di decisioni (istruzioni condizionali) presenti nel programma, le formule per il numero ciclomatico, rispettivamente per un programma monolitico e per un programma composto di p sottoprogrammi, diventano

$$v = d + 1$$

$$v = d + p$$

C.3.2 I criteri di Weyuker

La ricercatrice Elaine Weyuker ha proposto una serie di criteri che dovrebbero essere soddisfatti dalle metriche di complessità. Sebbene questi criteri siano piú che altro di interesse teorico e siano oltretutto controversi, sono un utile esempio del tipo di problemi che devono essere affrontati sia quando si inventa una metrica per qualche proprietà del software, sia quando vogliamo applicarne una già esistente, e dobbiamo quindi valutarne l'adeguatezza.

Nel séguito, P e Q denotano programmi o parti di programmi, $c()$ una misura di complessità, '$\equiv$' l'equivalenza funzionale, e ';' la concatenazione.

1. La misura di complessità non deve essere troppo grossolana (1).

   $$\exists P, Q \;\; c(P) \neq c(Q)$$

   (deve esistere almeno una coppia di programmi con complessità diverse).
2. La misura di complessità non deve essere troppo grossolana (2). Dato un numero non negativo $\bar{c}$, esiste solo un numero finito di programmi aventi complessità $\bar{c}$.
3. La misura di complessità non deve essere troppo fine.

   $$\exists P, Q \;\; c(P) = c(Q)$$

   (deve esistere almeno una coppia di programmi con la stessa complessità).
4. La complessità deve essere indipendente dalla funzionalità.

   $$\exists P, Q \;\; P \equiv Q \wedge c(P) \neq c(Q)$$

   (la misura deve poter discriminare due programmi che calcolano la stessa funzione).
5. La complessità deve essere monotòna rispetto alla concatenazione.

   $$\forall P, Q \;\; c(P) \leq c(P;Q) \wedge c(Q) \leq c(P;Q)$$

   (un programma è piú complesso di ciascuna sua parte).
6. Il contributo di una parte del programma alla complessità totale deve poter dipendere dal resto del programma.

   $$\exists P, Q, R \;\; c(P) = c(Q) \wedge c(P;R) \neq c(Q;R)$$

   $$\exists P, Q, R \;\; c(P) = c(Q) \wedge c(R;P) \neq c(R;Q)$$

   (due parti di programma di uguale complessità, concatenate ad una terza, possono dar luogo a programmi di complessità diversa).

7. La complessità deve dipendere dall'ordine in cui sono scritte le istruzioni.

$$\exists P, Q \text{ tali che } P \text{ sia una permutazione di } Q \text{ e } c(P) \neq c(Q)$$

8. La complessità non deve dipendere dalla scelta dei nomi di variabili e sottoprogrammi.

$$\forall P, Q \text{ tali che } P \text{ rinòmini } Q, c(P) = c(Q)$$

9. La complessità totale del programma può superare la somma delle complessità delle parti componenti.

$$\exists P, Q \;\; c(P) + c(Q) \leq c(P; Q)$$

La proprietà 6 fa sí che una misura di complessità tenga conto delle interazioni fra parti di programmi: due parti di programma di uguale complessità possono interagire in modo diverso col contesto in cui sono inseriti. Una misura che soddisfi questa proprietà, insieme alla proprietà 7, non è additiva, cioè non è possibile ricavare la complessità di un programma sommando le complessità delle parti componenti. Alcuni ricercatori ritengono, al contrario, che le misure di complessità debbano essere additive.

La proprietà 7 permette di tener conto dell'ordinamento delle istruzioni di un programma, e in particolare di distinguere la complessità in base all'annidamento dei cicli. Il numero ciclomatico non rispetta questa proprietà.

La proprietà 8 rispecchia il fatto che si vuole considerare la complessità come un attributo strutturale ed interno, distinto dalla comprensibilità (attributo di carattere psicologico ed esterno). Evidentemente, un programma che usa identificatori significativi è piú comprensibile di un programma ottenuto sostituendo gli identificatori con sequenze casuali di lettere, ma la struttura dei due programmi è identica e quindi ugualmente complessa.

C.4 Punti funzione

I *Punti Funzione* (*Function Points*) sono una metrica per i requisiti, e permettono, a partire dai requisiti utente di ottenere una valutazione preliminare dell'impegno richiesto dal progetto. Piú precisamente, i FP sono un numero legato al tipo ed al numero di funzioni richieste al sistema, ed a varie caratteristiche del sistema stesso. Un numero di FP piú alto corrisponde ad un sistema piú complesso e costoso.

Il metodo di calcolo prevede che un sistema possa offrire cinque tipi di funzioni. Dall'analisi dei requisiti bisogna trovare quante funzioni di ciascun tipo sono richieste, e valutare la complessità delle operazioni per ciascun tipo di funzione assegnandogli un peso numerico secondo la seguente tabella, che elenca i tipi di funzione previsti (contraddistinti da un indice i) con i pesi corrispondenti a tre livelli di complessità:

i	Tipi di funzione	Complessità		
		bassa	media	alta
1	File logici interni	7	10	15
2	File di interfaccia esterni	5	7	10
3	Ingressi esterni	3	4	6
4	Uscite esterne	4	5	7
5	Richieste esterne	3	4	6

I *file logici* sono gli insiemi di dati che devono essere mantenuti internamente dal sistema (ciascuno dei quali può corrispondere ad uno o piú file fisici, a seconda delle successive scelte di progetto che ne definiscono l'implementazione). I *file di interfaccia* sono i file scambiati con altri programmi, gli ingressi e le uscite sono le informazioni distinte fornite e, rispettivamente, ottenute dall'utente, le *richieste* sono le interrogazioni in linea che richiedono una risposta immediata dal sistema.

Alle caratteristiche del sistema (*General System Characteristics*), quali la riusabilità o la presenza di elaborazioni distribuite, che influenzano la complessità del progetto, viene assegnato un grado di influenza, cioè una valutazione dell'importanza di ciascuna caratteristica su una scala da zero a cinque. Le caratteristiche prese in considerazione dal metodo sono le quattordici mostrate nella tabella seguente, contraddistinte dall'indice j:

j	Caratteristiche del sistema
1	Riusabilità
2	Trasmissione dati
3	Elaborazioni distribuite
4	Prestazioni
5	Ambiente usato pesantemente
6	Ingresso dati in linea
7	Ingresso dati interattivo
8	Aggiornamento in tempo reale dei file principali
9	Funzioni complesse
10	Elaborazioni interne complesse
11	Facilità d'istallazione
12	Facilità di operazione
13	Siti molteplici
14	Modificabilità

La valutazione del grado d'influenza delle caratteristiche avviene sulla seguente scala:

Grado d'influenza	
0	Nullo
1	Scarso
2	Moderato
3	Medio
4	Significativo
5	Essenziale

Il numero di FP si calcola dalla seguente formula:

$$FP = (\sum_{i=1}^{5} N_i W_i)(0.65 + 0.01 \sum_{j=1}^{14} D_j)$$

dove N_i è il numero di funzioni di tipo i, W_i è il rispettivo peso, e D_j è il grado d'influenza della j-esima caratteristica.

C.5 Stima dei costi

Il costo di sviluppo di un sistema è ovviamente un parametro fondamentale per la pianificazione economica del lavoro. È importante poter prevedere i costi con ragionevole accuratezza, e a questo scopo sono stati proposti vari modelli che calcolano una stima dei costi a partire da alcuni parametri relativi al sistema da realizzare ed all'organizzazione che lo deve sviluppare.

Il costo di un sistema viene generalmente riassunto in due fattori: la quantità di lavoro, cioè l'impegno di risorse umane, ed il tempo di sviluppo. L'impegno di risorse umane viene di solito quantificato in termini di *mesi-uomo*, cioè del tempo necessario ad una persona per eseguire tutto il lavoro, ovvero il numero di persone necessarie a fare tutto il lavoro nell'unità di tempo. È bene ricordare che questa misura è soltanto convenzionale, ed evitare l'errore di credere che tempo di sviluppo e personale siano grandezze interscambiabili. Non è possibile dimezzare il tempo di sviluppo raddoppiando l'organico, poiché nel tempo di sviluppo entrano dei fattori, quali il tempo di apprendimento ed i tempi di comunicazione all'interno del gruppo di lavoro, che crescono all'aumentare del numero di partecipanti. Il tempo di sviluppo, quindi, viene di solito calcolato separatamente dalla quantità di lavoro. Fissati questi due fattori, si può quindi valutare il numero di persone da impiegare (*staffing*).

C.5.1 Il modello COCOMO

Il *COCOMO* (*COnstructive COst MOdel*) è un metodo di stima dei costi basato su rilevazioni statistiche eseguite su un certo numero di progetti di grandi dimensioni sviluppati presso una grande azienda produttrice di software. Esistono tre versioni del modello (*base*, *intermedio* e *avanzato*) di diversa accuratezza.

Ciascuna delle tre versioni prevede che le applicazioni vengano classificate in tre categorie:

organic: applicazioni semplici, di dimensioni limitate, di tipo tradizionale e ben conosciuto.
semi-detached: applicazioni di complessità media (software di base).
embedded: applicazioni complesse con requisiti rigidi di qualità ed affidabilità (sistemi in tempo reale).

Il principale dato d'ingresso del metodo è la dimensione del codice espressa il migliaia di linee di codice (KDSI), che deve essere stimata indipendentemente. Altre informazioni vengono usate per raffinare i risultati del calcolo, come vedremo fra poco.

Il modello base

Se S è la dimensione del codice in KDSI, la quantità di lavoro M in mesi-uomo ed il tempo di sviluppo T in mesi sono dati dalle formule:

$$M = a_b S^{b_b}$$

$$T = c_b M^{d_b}$$

dove i coefficienti a_b, b_b, c_b, e d_b si ricavano dalla seguente tabella:

Tipo dell'applicazione	a_b	b_b	c_b	d_b
Organic	2.4	1.05	2.5	0.38
Semi-detached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

Il metodo permette di calcolare la ripartizione della quantità di lavoro e del tempo di sviluppo fra le diverse fasi del ciclo di vita. Il COCOMO presuppone un ciclo a cascata articolato nelle quattro fasi di *pianificazione ed analisi*, *progetto*, *sviluppo*, ed *integrazione e test*. I costi si riferiscono alle ultime tre fasi, e la loro ripartizione fra queste fasi si ottiene da due tabelle (una per la quantità di lavoro ed una per il tempo di sviluppo) che dànno la rispettiva percentuale secondo il tipo e la dimensione dell'applicazione. Le stesse tabelle forniscono anche la percentuale relativa al costo aggiuntivo della fase di pianificazione ed analisi.

La ripartizione percentuale della quantità di lavoro fra le fasi del ciclo di vita si ricava dalla tabella seguente (P & A: pianificazione ed analisi; I & T: integrazione e test):

Tipo dell'applicazione	Fase	Dimensione (KDSI) 2	8	32	128	512
Organic	P & A	6	6	6	6	
	Progetto	16	16	16	16	
	Sviluppo	68	65	62	59	
	I & T	16	19	22	25	
Semi-detached	P & A	7	7	7	7	7
	Progetto	17	17	17	17	17
	Sviluppo	64	61	58	55	52
	I & T	19	22	25	28	31
Embedded	P & A	8	8	8	8	8
	Progetto	18	18	18	18	18
	Sviluppo	60	57	28	31	34
	I & T	22	25	28	31	34

La ripartizione del tempo di sviluppo si ricava dalla tabella seguente:

Tipo dell'applicazione	Fase	Dimensione (KDSI) 2	8	32	128	512
Organic	P & A	10	11	12	13	
	Progetto	19	19	19	19	
	Sviluppo	63	59	55	51	
	I & T	18	22	26	30	
Semi-detached	P & A	16	18	20	22	24
	Progetto	24	25	26	27	28
	Sviluppo	56	52	48	44	40
	I & T	20	23	26	29	32
Embedded	P & A	24	28	32	36	40
	Progetto	30	32	34	36	38
	Sviluppo	48	44	40	36	32
	I & T	22	24	26	28	30

Il modello intermedio

Nel modello intermedio si calcola dapprima la quantità di lavoro, detta *nominale*, che viene poi corretta secondo una formula che tiene conto di quindici coefficienti relativi ad altrettanti attributi dell'applicazione, del sistema di calcolo, del personale e dell'organizzazione del progetto.

La formula della quantità nominale di lavoro è la seguente:

$$M_n = a_i S^{b_i}$$

dove a_i e b_i si ottengono da questa tabella:

Tipo dell'applicazione	a_i	b_i
Organic	3.2	1.05
Semi-detached	3.0	1.12
Embedded	2.8	1.20

I coefficienti di correzione c_j vengono ottenuti dalla tabella seguente:

j	Attributo	Molto basso	Basso	Normale	Alto	Molto alto	Extra alto
1	RELY affidabilità	0.75	0.88	1.0	1.15	1.40	
2	DATA dimens. database		0.94	1.0	1.08	1.16	
3	CPLX complessità	0.70	0.85	1.0	1.15	1.30	1.65
4	TIME req. di efficienza		1.0	1.11	1.30	1.66	
5	STOR req. di memoria		1.0	1.06	1.21	1.56	
6	VIRT variabil. ambiente		0.87	1.0	1.15	1.30	
7	TURN tempo di risposta	0.87	1.0	1.07	1.15		
8	ACAP esper. analisti	1.40	1.19	1.0	0.86	0.71	
9	AEXP esper. applicazione	1.29	1.13	1.0	0.91	0.82	
10	PCAP esper. programmatori	1.42	1.17	1.0	0.86	0.70	
11	VEXP conosc. ambiente	1.21	1.10	1.0	0.90		
12	LEXP conosc. linguaggio	1.21	1.10	1.0	0.90		
13	MODP tecniche moderne	1.24	1.10	1.0	0.91	0.82	
14	TOOL strumenti software	1.24	1.10	1.0	0.91	0.83	
15	SCED scadenze per lo svil.	1.24	1.10	1.0	0.91	0.83	

La quantità di lavoro corretta M viene quindi calcolata dalla formula:

$$M = M_n \prod_{j=1}^{15} c_j$$

Il tempo di sviluppo si ottiene come nel modello base.

Il modello intermedio comprende una procedura per ripartire la quantità di lavoro fra i componenti dell'applicazione, cosí riassumibile:

1. Si stimano le dimensioni $S^{(k)}$ di ciascun componente (che identifichiamo con un indice k), da cui si ottiene la dimensione complessiva $S = \sum_k S^{(k)}$.
2. Si calcola la quantità nominale di lavoro complessiva M_n secondo la procedura già vista.
3. Si calcola la produttività nominale $p = S/M_n$, cioè il numero di linee di codice che dovrà essere prodotto per persona e per unità di tempo.
4. Si calcola la quantità nominale di lavoro $M_n^{(k)}$ di ciascun componente: $M_n^{(k)} = S^{(k)}/p$.
5. Si calcola la quantità corretta di lavoro $M^{(k)}$ di ciascun componente usando i rispettivi coefficienti correttivi: $M^{(k)} = M_n^{(k)} \prod_j c_j^{(k)}$.

Il modello avanzato

Il modello COCOMO avanzato introduce dei coefficienti correttivi anche nel calcolo della ripartizione dei costi fra le fasi di sviluppo. Inoltre, le stime si basano su un modello dell'applicazione piú articolato, che prevede un'architettura strutturata gerarchicamente in sottosistemi e moduli. Per ulteriori informazioni sul modello avanzato si faccia riferimento alla letteratura relativa.

Valutazione del metodo

Da alcune esperienze di applicazione del metodo COCOMO, risulta che è in grado di stimare i costi con un errore inferiore al 20% nel 25% dei casi per il modello base, nel 68% dei casi per il modello intermedio, e nel 70% dei casi per il modello avanzato.

Nel valutare l'applicabilità del metodo, bisogna tener presente che le formule e le tabelle sono state ottenute per mezzo di analisi statistiche su progetti con queste caratteristiche:

1. applicazioni generalmente complesse e critiche;
2. ciclo di vita a cascata;
3. requisiti stabili;
4. attività di sviluppo svolte in parallelo da piú gruppi di programmatori;
5. elevata maturità dell'organizzazione produttrice e competenza dei progettisti.

Un'organizzazione che voglia applicare questo metodo di stima deve quindi interpretare i risultati tenendo conto delle differenze fra la propria situazione effettiva (dal punto di vista dell'applicazione specifica e dell'ambiente di sviluppo) e quella presupposta dal modello. In particolare, bisogna tener conto degli aggiornamenti delle tabelle che vengono eseguiti e pubblicati periodicamente in base alle esperienze piú recenti. Inoltre è opportuno che le organizzazioni produttrici raccolgano ed elaborino le informazioni relative ai propri progetti, in modo da poter adattare il metodo alla propria situazione.

C.5.2 Il modello di Putnam

Nel modello di Putnam la dimensione L del programma (in LOC), la quantità di lavoro K (in anni–uomo), ed il tempo di sviluppo T_d (in anni) sono legate dalla formula

$$K = \frac{L^3}{C_k^3 t_d^4}$$

dove la *costante tecnologica* C_k è una caratteristica dell'organizzazione produttrice, valutata in base all'analisi dei suoi dati storici, che tiene conto della qualità delle metodologie applicate.

Appendice D

Gestione del processo di sviluppo

La gestione del processo di sviluppo è un argomento ampio che richiede di essere trattato da vari punti di vista, quali l'organizzazione aziendale, l'economia e la sociologia. Rinunciando ad affrontare tutte queste problematiche, alcune delle quali sono oggetto di altri corsi, nel séguito ci limiteremo ad accennare ad alcuni strumenti di uso comune, ed infine introdurremo il problema della gestione delle configurazioni.

D.1 Diagrammi WBS

I diagrammi di Work Breakdown Structure (struttura della suddivisione del lavoro) descrivono la scomposizione del progetto in base agli elementi funzionali del prodotto (WBS funzionale, fig. D.1) oppure in base alle attività costituenti lo sviluppo (WBS operativa, fig. D.2). Le unità di lavoro definite dalla WBS, dette task, sono raggruppate in un insieme di *work packages*. Ciascun work package definisce un insieme di task in modo abbastanza dettagliato da permettere ad un gruppo di persone di lavorarci indipendentemente dal resto del progetto. Per ogni work package bisogna stabilire le date di inizio e fine ed una stima del costo (*effort*) in termini di lavoro (mesi-persona) ed eventualmente di denaro (*budget*).

D.2 Diagrammi PERT

I diagrammi PERT (*Program Evaluation and Review Technique*) rappresentano i legami di precedenza fra le diverse attività del progetto. La dipendenza piú comune è quella di tipo "finish-to-start", in cui una attività può terminare solo dopo che un'attività precedente è terminata. Altre relazioni sono "start-to-start", in cui un'attività può iniziare insieme ad un'altra o dopo, e "finish-to-finish", in cui un'attività può terminare insieme ad un'altra o dopo. La figura D.3 mostra un ipotetico diagramma PERT con relazioni "finish-to-start", dove si suppone che il work package relativo al generatore di codice comprenda fra i propri task la definizione dei formati intermedi fra il parser ed il generatore e fra il generatore

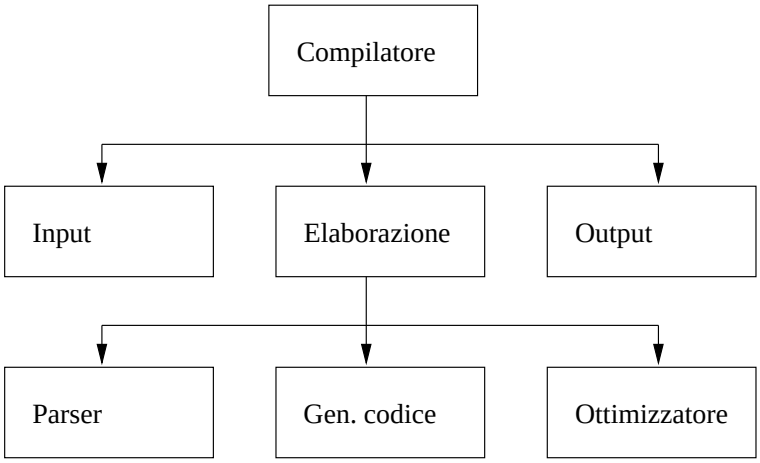


Figura D.1: WBS funzionale

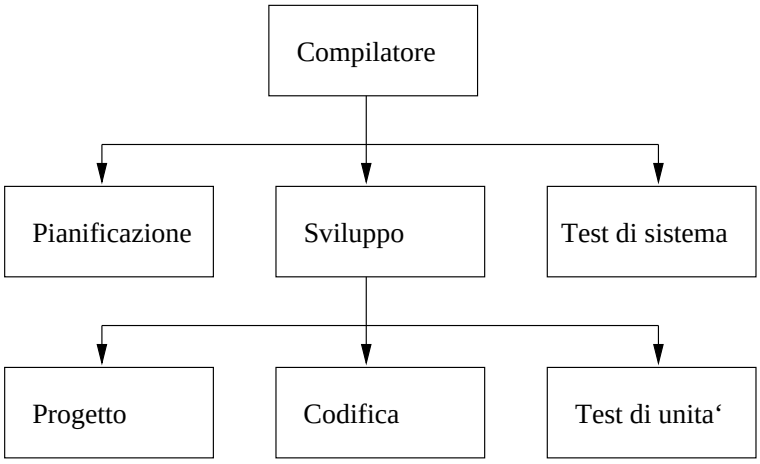


Figura D.2: WBS operativa

e l'ottimizzatore, per cui il completamento del generatore di codice deve precedere le attività di sviluppo del parser e dell'ottimizzatore.

A seconda dei legami di precedenza, può accadere che alcune attività possano essere ritardate entro un certo limite senza che il loro ritardo si ripercuota sul tempo complessivo di sviluppo. Questo margine di tempo si chiama *slack time*. Se nel grafo esiste una sequenza di attività senza slack time, cioè tali che un ritardo di ciascuna di esse porti ad un ritardo di tutto il progetto, la sequenza è un *cammino critico*.

D.3 Diagrammi di Gantt

I diagrammi di Gantt rappresentano ciascuna attività con una linea o una barra orizzontale parallela ad un asse dei tempi, e permette quindi di osservare immediatamente le date di inizio e fine, le durate, ed il parallelismo fra le varie attività (fig. D.4). Possono essere

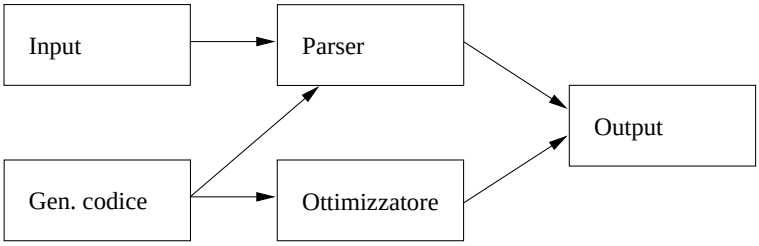


Figura D.3: Diagramma PERT

integrati dalle relazioni di precedenza viste nei diagrammi PERT. Nei diagrammi vengono messi in evidenza i *milestone*, cioè alcuni eventi particolarmente importanti, quali revisioni del lavoro e produzione di deliverable.

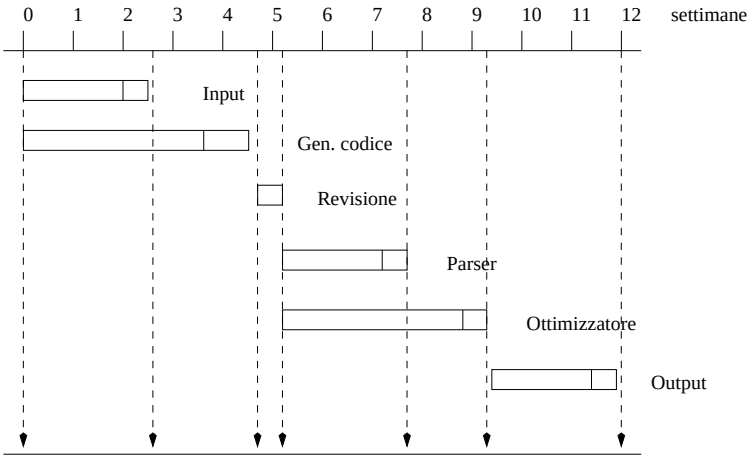


Figura D.4: Diagramma di Gantt

D.4 Esempio di documento di pianificazione

Riportiamo un breve estratto da un documento reale di pianificazione. Si tratta di un sistema embedded, e in particolare di un simulatore di veicoli capace di riprodurre gli stimoli meccanici, visivi ed acustici provati dal guidatore; il documento di pianificazione si riferisce al sistema completo, e non solo alla componente software. Il progetto viene svolto da un consorzio di quattro organizzazioni, che chiameremo AAA, BBB, etc., diretto da un comitato di gestione (*Steering Committee*). Il progetto è finanziato in parte dalla Comunità Europea (EC).

Il documento di pianificazione comprende un capitolo di introduzione ed analisi, che descrive la struttura del progetto in termini di workpackages e di fasi di sviluppo, e quindi descrive analiticamente i singoli workpackages ed i rispettivi task. L'estratto che riproduciamo viene da questo capitolo, e comprende l'introduzione, la descrizione del primo workpackage, e la descrizione del primo dei suoi task.

Il documento originale comprende anche tabelle che riassumono la ripartizione del lavoro fra i partner, diagrammi PERT e di Gantt, e la definizione delle politiche relative alla conformità (*compliance*) agli standard internazionali.

Workplan, deliverables

Introduction and analysis

The duration of the project is 36 months.

The project is organized in 10 workpackages and it is divided in 3 sequential phases:

1. user requirements and functional specification of the simulator;
2. design, implementation and testing of the main subsystems;
3. integration, final testing and user evaluation.

The first phase is composed by three work packages: WP1, WP2 and WP3, and its duration is 9 months. During this phase, all the high level requirements for the simulator and the virtual scenario are defined. The mathematical models of the vehicle and of the acoustic rendering are chosen. Finally, the global architecture of the simulator is designed and the functional requirements of the main components are specified.

The second phase is composed by five workpackages: WP4, WP5, WP6, WP7 and WP8, and its duration is 19 months. Each work package of this phase is devoted to one of the main subsystems of the simulator: the mechanical frame, the computer system for the mathematical model, the real-time system for the simulator control, the graphical and the acoustic subsystems. During this phase, each subsystem is designed, implemented and functionally tested.

The final phase is composed by the work package WP9 and its duration is 8 months. During this phase, all the subsystems are integrated in order to build the simulator prototype. The integration and the field tests are executed and the prototype is evaluated.

During the progress of the project it might be necessary to modify the workplan, both for technical reasons and for taking into account the feedback coming from market analysis. Any change in the workplan will be first agreed within the Steering Committee, and afterwards presented to the EC for approval. Each partner should update its activities in agreement with the consortium.

The activity of project management has to be performed during all phases.

The information dissemination will be started after the first phase.

These activities together are included in work package WP10.

There are 4 planned milestones during the project:
M0 effective commencement date T0.
M1 after 9 months. End of the specification phase and start of the implementation. All the modules are defined and their requirements specified.
M2 after 19 months. End of detailed design and beginning of the implementation phase. Checkpoint for the task of design and implementation of the five main subsystems.
M3 after 28 months. End of the implementation phase and start of the system integration. All the main subsystems are implemented and tested.
M4 end of the project. The simulator prototype is operational.

One critical path has raised up during the planning in correspondence with the last project activity. This is due to the low flexibility of the activity of field testing .... The easiest alternative action consists in increasing the project duration of at least one month .... Another alternative action is the overlapping of the "Integration test" task with the integration activity itself in such a way to provide an easy management of possible feedback from testing activities.

WP ID	WORKPACKAGE	PARTNER
WP1	Simulator User Requirements Definition	AAA
WP2	Mathematical Model Definition	AAA
WP3	Global architecture and Subsystem Specification	BBB
WP4	Mechanical Subsystem: Design, Implem.n and Test	BBB
WP5	Computational Subsystem: Design, Implem.n and Test	CCC
WP6	Real-time Subsystem: Design, Implem.n and Test	BBB
WP7	Graphical Subsystem: Design, Implem.n and Test	DDD
WP8	Acoustic Subsystem: Design, Implem.n and Test	DDD
WP9	Simulator Integration and Test	BBB
WP10	Project Management	AAA

WP1 Simulator User Requirements Definition [AAA]

Objectives: To define the specifications of the simulator from the user's point of view. This will be done first by a state-of-the art-analysis of existing simulators ... (task T1.1). A simultaneous market analysis will provide the feedback necessary to define a product suitable for a commercial exploitation. After that, the simulator functionality will be defined (task T1.2), and the virtual scenario will be outlined (task T1.3). Finally, specifications will be given (task T1.4), and the field test to be carried out at the end ... will be planned (Task T1.5).

Task T1.1 State-of-the-art of simulator systems [AAA]

Baselines: RP 1.1 State-of-the-art of simulator systems report
Duration: 1 month
Effort: AAA: 2.0, BBB: 1.0
Links: output to T1.2, T1.3, T1.6
Objectives: To gain a deeper knowledge about the state-of-the-art in simulator systems, both from the technical and the commercial point of view.
Approach: The various types of simulators ... will be analysed. Existing patents will be researched, as well as any other documentation available. If necessary, the manufacturers will be interviewed. The potential market will be estimated, and the potential customers will be identified.
Activities: Market analysis [AAA].
Overview on the existing simulators, with special attention on the control strategies related to inertial feedback [AAA].
Analysis of mechanical actuation solutions and computer architectures of existing simulator systems [BBB].
Risks: The available literature may not be sufficient for the proposed objective, and the interviewed manufacturers may not be well disposed to give the requested information, since this might be regarded as industrial know-how. The partners could propose to sign an agreement whereby they commit themselves to use the information only for the project, and not to transfer them to any third party. ...

È utile osservare che nell'introduzione c'è un paragrafo riguardante la possibilità di modificare il piano di lavoro, ed accenna implicitamente (senza specificarle) a delle procedure di modifica, indicando le strutture ed organizzazioni competenti (comitato di gestione e Comunità Europea). Viene inoltre messo in evidenza un cammino critico e vengono proposte delle azioni da prendere nel caso che il progetto subisca ritardi.

Ogni workpackage è affidato ad un partner, come risulta dalla tabella, che ha il ruolo di coordinatore per tale workpackage, a cui contribuiscono però anche gli altri partner.

Quindi la descrizione di ogni workpackage, di ogni task e di ogni attività entro i task specifica il partner responsabile. In particolare, La descrizione di ciascun task ha una voce (*Effort*) che specifica il contributo in mesi-uomo di ciascuna organizzazione.

Nella descrizione dei singoli task, la prima voce (*Baselines*) definisce il "semilavorato" che deve essere prodotto dal task. Nell'esempio riportato, si tratta di un rapporto, identificato dalla sigla RP 1.1 e dal titolo "State-of-the-art ...". La voce *Links* índica i rapporti di dipendenza fra i vari task, fornendo quindi le informazioni caratteristiche dei diagrammi PERT. Nell'esempio, vediamo che i task T1.1, T1.3 e T1.6 dipendono dal task T1.1. Infine, osserviamo che per ogni task vengono individuati i rischi associati e proposte eventuali soluzioni.

D.5 Gestione delle configurazioni

La gestione delle configurazioni si riferisce all'identificazione, l'organizzazione ed il controllo delle diverse versioni del prodotto, dei suoi componenti, della documentazione, e di tutte le informazioni relative allo sviluppo (per esempio, opzioni di compilazione, dati e risultati dei test, ...). Un insieme di elementi considerati come un'unità dal punto di vista della gestione delle configurazioni è un *configuration item*.

Piú precisamente, ogni versione del prodotto è individuata da un insieme di unità di configurazione (o moduli), cioè da una *configurazione*. Organizzare e controllare le versioni del prodotto significa controllare la distribuzione e la modifica degli elementi di configurazione, registrare e riferire lo stato degli elementi di configurazione e le relative richieste di modifica, e verificare la correttezza e completezza degli elementi di configurazione (ANSI/IEEE Std. 729-1983). In pratica, si tratta di far sí che ogni persona che debba lavorare sul prodotto, su un suo componente o su qualunque altro artefatto del processo di sviluppo, in qualsiasi fase del ciclo di vita, abbia a disposizione la versione corrente, e che non si creino inconsistenze dovute all'esistenza contemporanea di versioni diverse.

La pianificazione del progetto richiede anche la produzione di un piano di gestione delle configurazioni.

Una *versione* è uno stadio dell'evoluzione di un elemento di configurazione. Diverse versioni di un elemento di configurazione possono differire perché rispondono a requisiti diversi (offrono diverse funzioni e/o hanno diverse interfacce) o perché hanno diverse implementazioni degli stessi requisiti. Ciascuna versione può appartenere a diverse configurazioni. Ciascun elemento di configurazione e ciascuna delle sue versioni deve avere un identificatore. I numeri di versione sono strutturati in modo da distinguere i cambiamenti piú importanti (versione/edizioni) da quelli di minore portata (revisioni). I cambiamenti piú importanti naturalmente richiedono procedure di approvazione.

Una *baseline* è un documento o prodotto, costituito da un insieme di elementi di configurazione, che è stato revisionato ed accettato formalmente. Una baseline è una base per gli sviluppi ulteriori.

Per permettere la gestione delle configurazioni, ogni modulo o documento deve contenere le informazioni necessarie, quali identificatore, autore, data, e storia dei cambiamenti. È fondamentale l'uso di strumenti software, come SCCS o RCS.

Appendice E

Qualità

La qualità è "*l'insieme delle caratteristiche di un'entità che conferiscono ad essa la capacità di soddisfare esigenze espresse ed implicite*", dove un'entità può essere, fra l'altro, un prodotto, un processo, un servizio, o un'organizzazione (ISO 8402). Alcuni aspetti (parziali) della qualità sono *idoneità all'uso*, *soddisfazione del cliente*, o *conformità ai requisiti*. Osserviamo che la conformità ai requisiti è solo un aspetto necessario, ma non sufficiente, della qualità: in un certo senso, la qualità è ciò che fa preferire un prodotto ad un altro che pure soddisfa gli stessi requisiti.

La *gestione per la qualità* è l'insieme delle attività di gestione che traducono in pratica la *politica per la qualità* di un'organizzazione, cioè gli obiettivi e gli indirizzi generali riguardanti la qualità.

Il *sistema qualità* è costituito dalla struttura organizzativa, le procedure, i processi, e le risorse necessarie ad attuare la gestione per la qualità.

E.1 Le norme ISO 9000

Le norme ISO 9000 definiscono un insieme di requisiti necessari ad assicurare che un processo di sviluppo fornisca prodotti della qualità richiesta in modo *consistente*, cioè predicibile e ripetibile.

Gli standard principali dal punto di vista dell'industria del software sono:

ISO 8402 Gestione per la qualità e di assicurazione della qualità. Terminologia.

ISO 9000-1 Regole riguardanti la conduzione aziendale per la qualità e l'assicurazione della qualità. Guida per la scelta e l'utilizzazione.

ISO 9000-3 Regole riguardanti la conduzione aziendale per la qualità e l'assicurazione della qualità. Guida per l'applicazione della ISO 9001 allo sviluppo, alla fornitura e alla manutenzione del software.

ISO 9001 Sistemi qualità. Modello per l'assicurazione della qualità nella progettazione, sviluppo, fabbricazione, installazione ed assistenza.

ISO 9004-1 Gestione per la qualità e del sistema qualità. Guida generale.

ISO 9004-2 Elementi di gestione per la qualità e del sistema qualità. Guida per i servizi.

Il "cuore" di questo sistema di norme è la ISO 9001, che dà le direttive per un sistema di qualità che copre tutto il ciclo di vita di qualsiasi tipo di prodotto industriale. Affiancate alla ISO 9001 si trovano le norme ISO 9002 e ISO 9003 che si riferiscono a sottoinsiemi del ciclo di vita.

La norma ISO 9000 propriamente detta, in tre parti, è una guida che dà indicazioni per la scelta fra le norme (ISO 9001, 9002 o 9003) da applicare, e sulla loro applicazione. In particolare, la 9000-1 illustra i concetti principali e introduce l'insieme delle norme. La ISO 9000-3 è una guida all'applicazione della 9001 nelle aziende informatiche. La necessità di una guida rivolta specificamente all'industria del software discende dal fatto che la 9001, pur essendo proposta come riferimento del tutto generale, è stata concepita inizialmente per l'industria manifatturiera, per cui molte delle sue indicazioni non sono applicabili direttamente all'industria informatica. La ISO 9000-3 fornisce un'interpretazione della 9001 adatta alle condizioni proprie dell'industria informatica.

La ISO 9004 è una specie di "manuale utente" per le 9001, 9002 e 9003, dà una descrizione piú ampia dei rispettivi requisiti e fornisce indicazioni riguardo alle possibili scelte delle aziende per soddisfare tali requisiti. In particolare, la ISO 9004-2 si riferisce alle aziende fornitrici di servizi. Questa norma è rilevante per l'industria del software, poiché le aziende informatiche offrono dei servizi oltre che dei prodotti.

È importante rilevare che le norme contengono soltanto dei *requisiti* per il sistema qualità, e non prescrivono i metodi da applicare per soddisfarli: tocca alle singole organizzazioni fare queste scelte.

E.1.1 La qualità nelle norme ISO 9000

La norma ISO 9000-3 tratta i seguenti aspetti della gestione per la qualità:

1. Responsabilità della direzione;
2. sistema qualità;
3. verifiche ispettive interne del sistema qualità;
4. azioni correttive;
5. riesame del contratto;
6. specifica dei requisiti del committente;
7. pianificazione dello sviluppo;
8. pianificazione della qualità;
9. progettazione e realizzazione;
10. prova e validazione;
11. accettazione;
12. duplicazione, consegna ed installazione;
13. manutenzione;
14. gestione della configurazione;
15. controllo della documentazione;

16. documenti di registrazione della qualità;
17. misure;
18. regole, pratiche e convenzioni;
19. strumenti e tecniche;
20. approvvigionamenti;
21. prodotti software inclusi;
22. addestramento;

Quasi tutti i requisiti prevedono che le relative attività si svolgano secondo procedure pianificate e documentate, e che i risultati conseguenti vengano registrati, analizzati e conservati. Un testo sulle norme ISO 9000 sintetizza questo approccio con la frase "*dire cosa si fa, fare quel che si dice, e documentare*".

I primi tre requisiti si riferiscono al sistema di qualità in sé, e stabiliscono che la direzione predisponga una struttura ad esso dedicata, fornita delle risorse e dell'autorità necessarie ad applicare la politica della qualità. Il sistema qualità deve essere documentato, e in particolare può essere prodotto e messo a disposizione del personale un *manuale della qualità* contenente le regole e procedure aziendali. Le attività relative all'assicurazione della qualità devono essere pianificate.

Altri requisiti si riferiscono alle varie fasi del ciclo di vita. Come già osservato, le norme non prescrivono determinati modelli di sviluppo o metodologie, ma chiedono che vengano scelti ed applicati quelli che l'organizzazione produttrice ritiene adeguati. L'importante è che tutto sia documentato e verificato.

L'ultimo gruppo di requisiti (da *gestione della configurazione* in poi) si riferisce alle attività di supporto.

E.2 Certificazione ISO 9000

Un'organizzazione che ha adeguato il proprio sistema qualità allo standard ISO 9000 può ottenere una *certificazione* in merito. In alcuni casi la certificazione ISO 9000 è indispensabile per ottenere certi contratti, particolarmente con enti pubblici. La certificazione è importante anche nei rapporti fra aziende private, anche perché le norme stesse richiedono che il fornitore si accerti della qualità dei prodotti ottenuti dai subfornitori. Quindi ogni azienda che si impegna ad adottare lo standard ISO 9000 è motivata a rivolgersi a subfornitori che abbiano anch'essi adottato lo standard.

La certificazione viene rilasciata da organismi di certificazione, pubblici o privati, che a loro volta devono essere *accreditati*, cioè autorizzati a certificare, da enti di accreditamento. L'organismo di accreditamento italiano è il SINCERT (SIstema Nazionale di accreditamento per gli organismi di CERTificazione). È possibile ottenere la certificazione da piú di un organismo di certificazione, ed anche da organismi stranieri.

Il certificatore prima di tutto analizza la documentazione fornita dall'organizzazione richiedente relativa al sistema qualità, e in séguito compie una *verifica ispettiva* (*audit*)

inviando sul posto degli ispettori, che visitano i vari reparti dell'organizzazione ed intervistano il personale. Se il risultato dell'ispezione è positivo, viene emesso il certificato, che è valido per tre anni. In questi tre anni devono essere svolte altre verifiche ispettive, meno estese della prima, per accertare che la conformità allo standard venga mantenuta e migliorata. Il numero di verifiche svolte dopo la certificazione dipende dal certificatore (può quindi essere un fattore nella scelta del certificatore da parte del richiedente), e va da una a quattro all'anno. Dopo tre anni è necessario un nuovo processo di certificazione.

Nel corso dell'ispezione si possono rilevare delle non conformità. In questo caso vengono stabilite delle azioni correttive che saranno verificate in séguito. È possibile, su iniziativa del richiedente, svolgere un'ispezione di prova prima della verifica ispettiva per la certificazione.

Le verifiche ispettive seguono la norma ISO 10011.

E.3 Il Capability Maturity Model

Il *Capability Maturity Model* (*CMM*) è un modello, sviluppato dal Software Engineering Institute della Carnegie Mellon University, per valutare e quindi migliorare la qualità dei processi di sviluppo del software.

Il modello permette di valutare la *maturità* di un'organizzazione produttrice di software, cioè la sua capacità di gestire i processi di sviluppo. Sono previsti cinque livelli di maturità:

1. **Iniziale** (*Initial*) Ogni nuovo progetto viene gestito *ad hoc*, con poca o punta pianificazione[1].
2. **Ripetibile** (*Repeatable*) Esiste una gestione del progetto, capace di applicare le esperienze dei progetti passati a nuovi progetti riguardanti applicazioni simili.
3. **Definito** (*Defined*) Vengono applicate metodologie e tecniche ben definite sia per il processo di sviluppo che per la sua gestione. Tutti i progetti usano versioni particolari di un unico processo standardizzato nell'àmbito dell'organizzazione.
4. **Gestito** (*Managed*) Viene applicato un piano di misura per il controllo della qualità del prodotto e del processo.
5. **Ottimizzante** (*Optimizing*) I risultati delle misure vengono analizzati con metodi quantitativi ed usati per migliorare continuamente il processo di sviluppo. È prevista l'introduzione di nuove metodologie e tecniche.

Il CMM prevede delle strategie di miglioramento per le organizzazioni che vogliono passare ad un livello superiore.

da 1 a 2 Introduzione di gestione di progetto, pianificazione, gestione della configurazione, assicurazione della qualità.

[1] "*Few processes are defined, and success depends on individual effort and heroics*". SEI, `http://www.sei.cmu.edu/technology/cmm/cmm.sum.html`

da 2 a 3 Capacità di adattamento a nuovi problemi, di cambiare i processi.
da 3 a 4 Raccolta di dati quantitativi per misurare le proprietà dei processi e dei prodotti.
da 4 a 5 Aggiornamento dei processi in base ai risultati misurati.

Non si può stabilire una corrispondenza univoca fra la certificazione ISO 9000 ed il livello di maturità secondo il CMM, però un'organizzazione certificata ISO 9000 tipicamente ha una maturità corrispondente al terzo livello del CMM.

Il numero di organizzazioni che hanno conseguito una maturità corrispondente al massimo livello del CMM è tuttora molto ridotto.

Appendice F

Moduli in C++ e in Ada

F.1 Incapsulamento e raggruppamento in C++

Nel linguaggio C++ l'incapsulamento e il raggruppamento si ottengono con i costrutti `class` e `namespace`.

F.1.1 Classi e interfacce

In questo corso si presuppone la conoscenza del linguaggio C++ almeno nei suoi aspetti fondamentali, per cui non ci soffermeremo sul concetto di *classe* come definito in questo linguaggio, che è molto vicino a quello definito nell'UML. Ricordiamo soltanto che in C++ le operazioni virtuali pure corrispondono alle operazioni astratte dell'UML, e che una classe con almeno un'operazione virtuale pura è una classe astratta.

Una classe senza membri dato, che abbia solo operazioni virtuali pure con visibilità pubblica, permette di esprimere in C++ il concetto di *interfaccia*. Ricordando che un'interfaccia UML può contenere dichiarazioni di attributi, osserviamo che un attributo di un'interfaccia UML può essere rappresentato in C++ per mezzo di una coppia di operazioni virtuali pure, una per assegnare un valore all'attributo, e una per leggerlo.

F.1.2 Namespace

Una dichiarazione `namespace` definisce uno spazio di nomi, offrendo un meccanismo di raggruppamento piú semplice e generale delle classi. Infatti una classe, oltre a raggruppare delle entità, definisce le proprietà comuni di un insieme di oggetti che si possono istanziare, mentre uno spazio di nomi si limita ad organizzare un insieme di definizioni. Uno spazio di nomi può contenere definizioni di qualsiasi genere, incluse classi ed altri spazi di nomi. Questo costrutto può essere usato per rappresentare, al livello di astrazione del codice sorgente, concetti architetturali piú astratti, come package e componenti.

Il seguente esempio mostra la dichiarazione di un `namespace` (contenuta in un file di intestazione `Linear.h`) che raggruppa due classi:

```
// Linear.h

namespace Linear {
    class Vector {
        double* p;
        int  size;
    public:
        enum { max = 32000 };
        Vector(int s);
        double& operator[](int i);
        // ...
    };

    class Matrix {
        // ...
    };
}
```

L'implementazione del modulo la cui interfaccia è definita dallo spazio di nomi `Linear` può essere definita in un altro file, `Linear.cc`:

```
// Linear.cc

#include "Linear.h"

using namespace Linear;

Vector::Vector(int s)
{
    p = new double[size = s];
}

double&
Vector::operator[](int i)
{
    return p[i];
}

// altre definizioni per Vector e Matrix
// ...
```

La direttiva `#include`, interpretata dal preprocessore, serve a includere il file `Linear.h`, rendendo cosí visibili al compilatore le dichiarazioni contenute in tale file. In questo caso, al modulo logico specificato dal `namespace Linear` corrisponde un modulo fisico costituito dall'unità di compilazione formata da `Linear.cc` e `Linear.h`. La direttiva `using namespace`, interpretata dal compilatore, dice che quando il compilatore trova dei nomi non dichiarati, deve cercarne le dichiarazioni nello spazio di nomi `Linear`. Questa direttiva, quindi, rappresenta approssimativamente la relazione ≪import≫ fra i package. Alternativamente all'uso della direttiva `using namespace` si possono usare delle dichiarazioni `using`, una per ciascun nome appartenente a `Linear` usato nell'unità di compilazione:

```
using Linear::Vector;
```

Oppure, si può qualificare esplicitamente ogni occorrenza di nomi appartenenti a `Linear`. Per esempio, la definizione del costruttore della classe `Vector` può essere scritta cosí:

```
int&
Linear::Vector::operator[](int i)
{
    // ...
}
```

Un modulo cliente deve includere il file di intestazione e usare i nomi in esso dichiarati, con le convenzioni linguistiche già viste:

```
// main.cc

#include <iostream>
#include "Linear.h"

using namespace Linear;

main()
{
    Vector v(4);

    v[0] = 0.0;
    // ...
}
```

La direttiva `using namespace` può essere usata per comporre spazi di nomi, come nel seguente esempio:

```
// Linear.h

namespace Linear {
    class Vector {
        // ...
    };

    class Matrix {
        // ...
    };
}
.............................
// Trig.h

namespace Trig {
    double sin(double x);
    double cos(double x);
    // ...
}
.............................
```

```
// Math.h

#include "Linear.h"
#include "Trig.h"

namespace Math {
    using namespace Linear;
    using namespace Trig;
}
```

F.1.3 Moduli generici in C++

In C++ i moduli generici si realizzano col meccanismo dei template, o classi modello. Di séguito mostriamo una possibile definizione di uno stack di elementi di tipo generico:

```
// Stack.h

template<class T>
class Stack {
    T* v;
    T* p;
    int size;
public:
    Stack(int s);
    ~Stack();
    void push(T a);
    T pop();
};

template<class T>
Stack<T>::Stack(int s)
{
    v = p = new T[size = s];
}

// altre definizioni
// ...
```

Questo modulo generico può essere usato in un modulo cliente come questo:

```
#include <iostream.h>
#include "Stack.h"

main()
{
    Stack<int> si(10);

    si.push(3);
    si.push(4);
    cout << si.pop() << endl << si.pop() << endl;
}
```

In C++, una classe generica viene istanziata implicitamente quando si dichiara un oggetto. Spetta all'ambiente di programmazione generare il codice nel modo opportuno (possibilmente senza duplicazioni).

Una classe generica può essere parametrizzata anche rispetto a valori costanti e a funzioni. Nell'esempio seguente la classe `VectOp` fornisce due operazioni su array di elementi generici: la funzione `elementwise()` applica a due array, elemento per elemento, un'operazione binaria generica `Binop()`, mentre la funzione `fold()` applica la stessa operazione, in sequenza, a tutti gli elementi di un array, "accumulando" il risultato (questo modo di applicare un'operazione agli elementi di una sequenza si chiama *folding* nel campo della programmazione funzionale). La funzione `elementwise()` è analoga alla somma vettoriale, mentre la `fold()` è analoga alla sommatoria di una successione finita di numeri. La classe `VectOp` è quindi generica rispetto a tre parametri: il tipo `T` degli elementi delle sequenze (rappresentate come array) su cui si opera, la dimensione `Size` delle sequenze, e l'operazione `Binop` da impiegare nei due algoritmi `elementwise()` e `fold()`[1].

```
// VectOp.h

template<class T, int Size, class T (*Binop)(class T, class T)>
class VectOp {
public:
    static void elementwise(T*, T*, T*);
    static void fold(T*, T&);
};

template<class T, int Size, class T (*Binop)(class T, class T)>
inline void VectOp<T,Size,Binop>::elementwise(T* a, T* b, T* r)
{
    for (int i = 0; i < Size; i++)
        r[i] = Binop(a[i], b[i]);
}

template<class T, int Size, class T (*Binop)(class T, class T)>
inline void VectOp<T,Size,Binop>::fold(T* a, T& r)
{
    for (int i = 0; i < Size; i++)
        r = Binop(r, a[i]);
}
```

Questa classe può essere usata come nel seguente esempio:

```
#include <iostream>
#include <string>
#include "VectOp.h"

string concatenate(string s1, string s2)
{
    return s1 + s2;
}
```

[1] Si confronti questo esempio con le funzioni modello `accumulate()`, `inner_product()`, `partial_sum()` e `adjacent_differences()` della libreria standard del C++.

```
main()
{
    const int SIZE = 3;
    VectOp<string, SIZE, concatenate> vo;

    string seq1[] = { "blue ", "white ", "green ", };
    string seq2[] = { "sky", "snow", "forest", };

    string res_seq[SIZE];
    vo.elementwise(seq1, seq2, res_seq);
    for (int i = 0; i < SIZE; i++)
        cout << res_seq[i] << endl;

    string res_string;
    vo.fold(seq1, res_string);
    cout << res_string << endl;
}
```

Osserviamo che la libreria standard del C++ (*Standard Template Library*, STL) offre numerose classi e algoritmi generici, che permettono di risolvere molti problemi di programmazione in modo affidabile ed efficiente.

F.1.4 Eccezioni in C++

In C++ un'eccezione viene sollevata mediante un'istruzione `throw`, che ha un operando di tipo qualsiasi. Uno handler è costituito da un'istruzione `catch`, formata dalla parola `catch` seguita da una dichiarazione di argomento formale (analogamente alle dichiarazioni di funzione) e da un'istruzione. L'eccezione sollevata da un'istruzione `throw` viene gestita da uno handler il cui argomento formale ha lo stesso tipo dell'operando della `throw`. Conviene definire delle classi destinate a rappresentare i diversi tipi di eccezione (non esistono eccezioni predefinite dal linguaggio, ma alcune eccezioni sono definite nella libreria standard). Nel seguente esempio vengono definite le classi `RangeExc` e `SizeExc`, che rappresentano rispettivamente le eccezioni sul valore dell'indice di un elemento e sulla dimensione del vettore. Le due classi sono vuote (cosa permessa dalla sintassi del linguaggio), poiché in questo caso non ci interessa associare informazioni alle eccezioni.

Nel file di intestazione della classe `Vector`, le eccezioni sollevate da ciascuna funzione vengono dichiarate esplicitamente per mezzo delle clausole `throw` (diverse dalle *istruzioni* `throw`):

```
// Vector.h

class Vector {
    double* p;
    int  size;
public:
    enum { max = 32000 };
    class RangeExc {};
    class SizeExc {};
```

```
    Vector(int s) throw (SizeExc);
    double& operator[](int i) throw (RangeExc);
    // ...
};
```

Le clausole `throw` vengono ripetute anche nelle definizioni, in modo che il compilatore possa verificare la coerenza fra dichiarazioni e definizioni:

```
// Vector.cc

#include "Vector.h"

Vector::Vector(int s) throw (SizeExc)
{
    if (s < 0 || max < s)
        throw SizeExc();
    p = new double[size = s];
}

double&
Vector::operator[](int i) throw (RangeExc)
{
    if (0 <= i && i < size)
        return p[i];
    else
        throw RangeExc();
}
```

In questo esempio gli operandi delle istruzioni `throw` sono i costruttori delle due classi: tali costruttori producono oggetti vuoti.

Le istruzioni la cui esecuzione può sollevare un'eccezione che vogliamo gestire vengono raggruppate in un blocco `try`. Questo blocco viene seguito dagli handler:

```
#include <iostream>
#include "Vector.h"

double
g(Vector v, int i)
{
    return v[i];
}

main()
{
                                  // eccezioni non gestite
    Vector v(4);

    for (int i = 0; i < 4; i++)
        v[i] = i * 2.3;
    try {                         // eccezioni catturate
        double x = g(v, 5);
        cout << x << endl;
    } catch (Vector::RangeExc) {
```

```
        cerr << "Range Exception" << endl;
    } catch (Vector::SizeExc) {
        cerr << "Size Exception" << endl;
    }
}
```

Se si verificano eccezioni nell'esecuzione di istruzioni esterne al blocco `try`, queste eccezioni vengono propagate lungo la catena delle chiamate finché non si trova un gestore. Se non ne vengono trovati, il programma termina.

Si possono associare informazioni alle eccezioni, usando classi non vuote:

```
class Vector {
    // ...
public:
    enum { max = 32000 };
    class RangeExc {
    public:
        int index;
        RangeExc(int i) : index(i) {};
    };
    class SizeExc {};
    Vector(int s) throw (SizeExc);
    double& operator[](int i) throw (RangeExc);
    // ...
};

double&
Vector::operator[](int i) throw (RangeExc)
{
    if (0 <= i && i < size)
        return p[i];
    else
        throw RangeExc(i);
}
```

L'informazione associata all'eccezione può quindi essere usata in uno handler, se questo ha un argomento formale che denota l'eccezione lanciata:

```
main()
{
    // ...
    try {
        // ...
    } catch (Vector::RangeExc r) {
        cerr << "bad index: " << r.index << endl;
    } catch (Vector::SizeExc) {
        // ...
    }
}
```

Le eccezioni possono essere raggruppate usando l'eredità: per esempio, in un modulo che esegue dei calcoli numerici si possono definire delle eccezioni che rappresentano errori aritmetici generici (`MathError`), e da queste si possono derivare eccezioni piú specifiche, come il traboccamento (`Ovfl`) o la divisione per zero (`ZeroDiv`).

```
class MathError {};
class Ovfl : public MathError {};
class ZeroDiv : public MathError {};
```

Un modulo cliente può trattare le eccezioni in modo differenziato:

```
try {
    // ...
} catch (Ovfl) {
    // ...
} catch (MathError) {
    // ...
} catch ( ... ) {
    // ...
}
```

Nell'ultimo handler, i punti di sospensione sono la sintassi usata per specificare che lo handler gestisce qualsiasi eccezione. Quando viene sollevata un'eccezione, il controllo passa al primo gestore, in ordine testuale, il cui argomento formale ha un tipo uguale a quello dell'eccezione, o un tipo piú generale. Da questa regola segue che gli handler piú generali devono seguire quelli piú specializzati, altrimenti questi ultimi non possono essere eseguiti.

F.2 Moduli in Ada

In Ada, i moduli sono rappresentati per mezzo dei *package*. Un package ha una specifica (interfaccia) ed un corpo (implementazione), che possono essere definiti in due file distinti e compilabili separatamente, come nel seguente esempio:

```
-- file StackADT_.ada
-- interfaccia

package StackADT is
    type Stack is private;
    procedure Push(S: in out Stack; E : in  Integer);
    procedure Pop (S: in out Stack; E : out Integer);
private
    type Table is array (1 .. 10) of integer;
    type Stack is
        record
            t : Table;
            index : integer := 0;
        end record;
end StackADT;

.............................
-- file StackADT.ada
-- implementazione

package body StackADT is
```

```
    procedure Push(S : in out Stack; E : in Integer) is
    begin
        S.index := S.index + 1;
        S.t(S.index) := E;
    end Push;

    procedure Pop(S : in out Stack; E : out Integer) is
    begin
        -- ...
    end Pop;
end StackADT;
```

In questo esempio si realizza un tipo di dato astratto che presenta la tipica interfaccia degli stack. L'implementazione si basa su due strutture dati: il vettore `Table` e il record (o struttura, nella terminologia del C++) `Stack`. Queste due strutture sono dichiarate nella parte privata della specifica. Il package `StackADT` può essere usato come segue:

```
-- file StackADTTest.ada

with StackADT; use StackADT;
procedure StackADTTest is
    i : INTEGER;
    s : Stack;
begin
    push(s, 1);
    push(s, 2);
    pop(s, i);
    -- ...
end StackADTTest;
```

La clausola `with` esprime la dipendenza di `StackADTTest` da `StackADT`. La clausola `use` rende visibili i nomi dichiarati nel package `StackADT`, permettendo di usarli senza qualificarli esplicitamente col nome del package.

F.2.1 Moduli generici

La specifica di un package generico inizia con la parola chiave `generic`, seguita dai nomi e tipi dei parametri:

```
generic
    Size : Positive;
    type Item is private;
package Stack is
    procedure Push(E : in  Item);
    procedure Pop (E : out Item);
    Overflow, Empty : exception;
end Stack;

package body Stack is
    type Table is array (Positive range <>) of Item;
```

```
    Space : Table(1 .. Size);
    Index : Natural := 0;

    procedure Push(E : in Item) is
    begin
        end if;
        Index := Index + 1;
        Space(Index) := E;
    end Push;

    -- altre definizioni
    -- ...
end Stack;
```

Questo package può essere usato nel modo seguente:

```
package Stack_Int is new Stack(Size => 200, Item => Integer);

Stack_Int.Push(7);
```

Anche in Ada un modulo può essere parametrizzato rispetto a delle funzioni. I parametri funzione di un package generico vengono dichiarati con la parola chiave `with`:

```
generic
    type ITEM is private;
    type VECTOR is array(POSITIVE range <>) of ITEM;
    with function SUM(X, Y : ITEM) return ITEM;
        -- questa e' la somma generica fra elem. di VECTOR
package VECT_OP is
    function SUM(A, B : VECTOR) return VECTOR;
        -- questa e' la somma fra oggetti VECTOR
    function SIGMA(A : VECTOR) return ITEM;
        -- questa e' la somma degli elementi di un VECTOR
end;
```

Questo package può essere istanziato in questo modo:

```
package INT_VECT is new VECT_OP(INTEGER, TABLE, "+");
```

F.2.2 Eccezioni

In Ada le eccezioni sollevate in un package possono essere dichiarate come nel seguente esempio:

```
package StackADT is
    type Stack is private;
    procedure Push(S: in out Stack; E : in  Integer);
    procedure Pop (S: in out Stack; E : out Integer);
```

```
    Full, Empty : exception;
private
    type Table is array (1 .. 10) of integer;
    type Stack is
        record
            t : Table;
            index : integer := 0;
        end record;
end StackADT;
```

L'istruzione `raise` viene usata per sollevare un'eccezione:

```
package body StackADT is
    procedure Push(S : in out Stack; E : in Integer) is
    begin
        if S.index >= 10 then
            raise Full;
        end if;
        -- ...
    end Push;

    procedure Pop(S : in out Stack; E : out Integer) is
    begin
        if S.index = 0 then
            raise Empty;
        end if;
        -- ...
    end Pop;
end StackADT;
```

Uno handler è una sequenza di istruzioni compresa fra la parola chiave `exception` e la fine del blocco cui appartiene lo handler. Generalmente uno handler è costituito da una sequenza di clausole `when`, che specificano le azioni da compiere quando viene sollevata un'eccezione. Nel seguente esempio, si suppone che la procedura `StackADTTest` possa sollevare l'eccezione `Full`:

```
procedure StackADTTest is
    i : INTEGER;
    s : Stack;
begin
    push(s, 1);
    push(s, 2);
    -- ...
exception
    when Full =>
        -- ...
end StackADTTest;
```

Il linguaggio fornisce delle eccezioni predefinite, fra cui, per esempio, quelle relative ad errori nell'uso di vettori, record, puntatori (`constraint_error`), e ad errori aritmetici (`numeric_error`). Le eccezioni predefinite

vengono sollevate implicitamente da dei controlli (*check*) eseguiti dal supporto run-time del linguaggio, che possono essere disabilitati. Per esempio, i check associati alle eccezioni `constraint_error` sono `access_check` (uso di puntatori), `index_check` (indici di array), ed altri ancora. Per disabilitare un check, si usa la direttiva `pragma`:

```
pragma SUPPRESS(INDEX_CHECK);
```

F.2.3 Eredità in Ada95

Accenniamo brevemente al modo in cui l'eredità è stata introdotta in Ada 95, la versione piú recente del linguaggio Ada.

I tipi che possono servire da base per l'eredità sono chiamati *tagged*. Il seguente esempio mostra la specifica di un package in cui viene definito un tipo base (tagged) `Figure` da cui vengono derivati i tipi `Circle`, `Rectangle` e `Square`. Quest'ultimo deriva da `Rectangle`.

```
package Figures is

    type Point is
        record
            X, Y: Float;
        end record;

    type Figure is tagged
        record
            Start : Point;
        end record;
    function Area (F: Figure) return Float;
    function Perimeter (F:Figure) return Float;
    procedure Draw (F: Figure);

    type Circle is new Figure with
        record
            Radius: Float;
        end record;
    function Area (C: Circle) return Float;
    function Perimeter (C: Circle) return Float;
    procedure Draw (C: Circle);

    type Rectangle is new Figure with
        record
            Width: Float;
            Height: Float;
        end record;
    function Area (R: Rectangle) return Float;
    function Perimeter (R: Rectangle) return Float;
    procedure Draw (R: Rectangle);

    type Square is new Rectangle with null record;

end Figures;
```

In questo esempio, i tipi `Circle` e `Rectangle` ereditano da `Figure` il campo `Start`, a cui aggiungono i campi propri dei tipi derivati. Le funzioni appartenenti all'interfaccia vengono ridefinite per ciascun tipo; osserviamo che l'oggetto su cui operano tali funzioni viene passato esplicitamente come parametro. Il tipo `Square` eredita da `Rectangle` senza aggiungere altri campi, come índica la clausola `null record`.

Il linguaggio Ada 95 permette anche di definire funzioni astratte, analogamente alle funzioni virtuali pure del C++.

Bibliografia

[1] ECSS-E-40B Draft 1. *Space engineering – Software*. ESA-ESTEC, 2002.

[2] ISO/IEC 12207:2008(E). *Systems and software engineering – Software life cycle processes — Ingénierie des systèmes et du logiciel – Processus du cycle de vie du logiciel*. ISO/IEC and IEEE, 2008.

[3] ESA PSS-05-0 Issue 2. *ESA Software Engineering Standards – issue 2*. European Space Agency, 1991.

[4] IAEA TRS 384. *Verification and Validation of Software Related to Nuclear Power Plant Instrumentation and Control*. IAEA, International Atomic Energy Agency, 1999.

[5] CEI EN 50128:2002-04. *Applicazioni ferroviarie, tranviarie, filotranviarie, metropolitane – Sistemi di telecomunicazione, segnalamento ed elaborazione – Software per sistemi ferroviari di comando e di protezione — Railway applications – Communications, signalling and processing systems – Software for railway control and protection systems*. CEI, Comitato elettrotecnico italiano, 2002. European standard.

[6] ISO 8807:1989. *Information processing systems – Open Systems Interconnection — LOTOS – A formal description technique based on the temporal ordering of observational behaviour*. ISO, Intenational Organization for Standardization, 1989.

[7] ARIANE 5 Flight 501 Failure — Report of the Inquiry Board. report, Ariane 501 Inquiry Board, 1996. Board chairman: J. L. Lions.

[8] J. Arlow and I. Neustad. *UML 2 and the Unified Process, Second Edition*. Addison-Wesley, 2005.

[9] Mordechai Ben-Ari. The bug that destroyed a rocket. *Journal of Computer Science Education*, 13(2):15–16, 1999.

[10] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks*, 14:25–59, 1987.

[11] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer-Verlag, second edition, 1984.

[12] Leonardo de Moura, Sam Owre, Harald Rueß, John M. Rushby, Natarajan Shankar, Maria Sorea, and Ashish Tiwari. SAL 2. In Rajeev Alur and Doron A. Peled, editors, *Computer Aided Verification, 16th International Conference, (CAV 2004)*, volume 3114 of *Lecture Notes in Computer Science*, pages 496–500. Springer, 2004.

[13] R. Fairley. *Software Engineering Concepts*. McGraw-Hill, 1985.

[14] R. W. Floyd. Assigning meaning to programs. In *Proceedings of the American Mathematical Society Symposia on Applied Mathematics*, volume 19, pages 19–32. AMS, 1967.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[16] C. Ghezzi, A. Fuggetta, S. Morasca, A. Morzenti, and M. Pezzè. *Ingegneria del Software – Progettazione, sviluppo e verifica*. Mondadori, 1991.

[17] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, 1991.

[18] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Ingegneria del Software – Fondamenti e principi*. Pearson Education Italia, 2004.

[19] Tom Gilb. *Principles of software engineering management*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[20] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, July(8):231–274, 1987.

[21] Michi Henning and Steve Vinoski. *Advanced CORBA Programming with C++*. Addison–Wesley, 1999.

[22] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–583, October 1969.

[23] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

[24] F. Jahanian and A. K. Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, SE-12(9):890–904, September 1986.

[25] Nancy Leveson. *Software: System Safety and Computers*, chapter Medical Devices: The Therac-25. Addison-Wesley, 1995.

[26] Nancy G. Leveson. The therac-25: 30 years later. *Computer*, 50(11):8–11, 2017.

[27] J.R. Levine, T. Mason, and D. Brown. *lex & yacc*. O'Reilly & Associates, 1992.

[28] H. Lichter, M. Schneider-Hufschmidt, and H. Züllighoven. Prototyping in industrial software projects – bridging the gap between theory and practice. *IEEE Trans. on Software Engineering*, 20(11), November 1994.

[29] R.C. Linger. Cleanroom process model. *IEEE Software*, 11(2):50–58, 1994.

[30] E. Mendelson. *Introduction to Mathematical Logic*. 3rd Edition, Van Nostrand, 1987.

[31] Bertrand Meyer. *Agile! - The Good, the Hype and the Ugly*. Springer, 2014.

[32] MIL-STD-498. *Software development and documentation*. US Department of Defense, 1994.

[33] MISRA-C:2004 — Guidelines for the use of the C language in critical systems. Technical Report MISRA-C:2004, Motor Industry Software Reliability Association, 1998.

[34] IAEA NS-G-1.1. *Software for Computer Based Systems Important to Safety in Nuclear Power plants*. IAEA, International Atomic Energy Agency, 2000.

[35] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, number 1102 in Lecture Notes in Computer Science, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[36] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053âĂŞ1058, dec 1972.

[37] D.L. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, SE-5(2):128–138, 1979.

[38] D.L. Parnas. Software aging. In *Proceedings of 16th International Conference on Software Engineering*, pages 279–287, 1994.

[39] A. Pope. *The CORBA Reference Guide*. Addison-Wesley, 1997.

[40] R. S. Pressman. *Software Engineering: a Practitioner's Approach*. McGraw-Hill, 2005.

[41] W. V. Quine. *Elementary Logic – Revised edition*. Harvard University Press, 1980.

[42] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[43] SEI CERT C Coding Standard — Rules for Developing Safe, Reliable, and Secure Systems. Technical report, Software Engineering Institute-Carnegie Mellon University, 2016.

[44] I. Sommerville. *Software Engineering*. Addison-Wesley, 1995.

[45] Friedrich von Henke, Cinzia Bernardeschi, Paolo Masci, and Hélène Waeselynck. *Testing, Verification and Validation*. ReSIST Courseware. `http://resist.isti.cnr.it/files/corsi/courseware_slides/testing.pdf`.

[46] J. Woodcock and J. Davies. *Using Z – Specification, Refinement, and Proof*. Prentice Hall, 1996.

Indice analitico