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 in Ingegneria informatica. Tutte le nozioni qui esposte, escluse le appendici, saranno materia di esame, incluse le parti non trattate a lezione. 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.

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:

software

> *"i programmi, le procedure, le regole e l'eventuale documentazione associata, e i dati relativi all'operatività di un sistema di elaborazione"*
> *– IEEE Standard Glossary of Software Engineering Terminology [5]*

ingegneria del software

> *"l'applicazione di un approccio sistematico, disciplinato, quantificabile, allo sviluppo, all'operatività, alla manutenzione ed al ritiro del software"*
> *– IEEE Standard Glossary of Software Engineering Terminology [5]*

> *"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 [11]*

È particolarmente importante, nelle due definizioni di "ingegneria del software", il concetto di sistematicità. La necessità di sottolineare l'importanza di questo concetto, che è data per scontata nelle altre discipline ingegneristiche, può essere meglio compresa se consideriamo la storia dell'informatica, schematizzata nelle tre fasi di *arte*, *artigianato*, e *industria*.

La prima fase, quella delle origini, è caratterizzata dal fatto che i produttori e gli utenti del software condividevano una formazione scientifica, per cui i programmatori avevano una certa familiarità con i problemi degli utenti, e questi ultimi erano in grado di capire il funzionamento dei calcolatori e quindi le esigenze dei programmatori. Anzi, accadeva spesso che utente e programmatore fossero la stessa persona. Un altro aspetto caratteristico di quel periodo è il fatto che spesso un'applicazione venisse prodotta *ad hoc* per un particolare problema e venisse abbandonata una volta soddisfatta quella particolare necessità. La produzione del software in quel periodo era quindi paragonabile ad un'arte, nel senso che era dominata dall'inventività individuale e da un'organizzazione del lavoro molto lasca (tralasciamo qui altri aspetti, pur importanti, legati alle tecniche di programmazione ed agli strumenti di sviluppo, in quel periodo assai limitati).

In una fase successiva, in seguito allo sviluppo delle applicazioni informatiche, estese dal campo scientifico a quello commerciale e amministrativo, i nuovi utenti del software hanno formazioni culturali diverse da quella dei programmatori. Questi ultimi devono quindi imparare ad affrontare problemi di tipo nuovo, ed a comunicare efficacemente con persone che non solo non hanno le conoscenze necessarie a capire gli aspetti tecnici della programmazione, ma non hanno neppure alcuna motivazione pratica per interessarsene. Inoltre, le applicazioni richieste hanno un peso economico sempre maggiore ed un ruolo sempre piú critico nelle organizzazioni che ne fanno uso. I programmatori assumono quindi una figura professionale distinta, ed il lavoro viene organizzato su gruppi di programmatori, che possono essere imprese autonome (*software houses*) o parti di organizzazioni (servizi di elaborazione

dati o di sviluppo del software). In questa fase c'è una maggiore strutturazione dell'attività di produzione, però rimangono l'approccio individualistico alla programmazione e la mancanza di strumenti teorici e di pratiche standardizzate. In questo senso si può parlare di artigianato, cioè di un'attività "pre-industriale", motivata economicamente ma sempre basata sulle capacità individuali e poco standardizzata.

L'inadeguatezza degli strumenti teorici e metodologici nella fase artigianale della produzione di software fece sentire i propri effetti nella famosa *crisi del software*, fra gli anni '60 e '70: la produzione del software non riusciva a tenere il passo con le richieste degli utenti e con i progressi dello hardware, aumentavano i costi di sviluppo, i ritardi nelle consegne, ed i malfunzionamenti, con un conseguente aumento dei costi di manutenzione. Si capí quindi che la via d'uscita era il passaggio da artigianato a industria: era cioè necessario (e lo è tuttora) usare nella produzione di software lo stesso approccio che si usa nelle industrie mature, come l'industria meccanica o l'industria elettronica. Nella conferenza di Garmisch [23] del 1968 (anno cruciale anche per altri aspetti) venne coniato il termine di "ingegneria del software".

La transizione da artigianato a industria è tuttora in corso: la disciplina dell'ingegneria del software è ancora in fase di crescita e lontana, in molti dei suoi settori, dal consolidamento di teorie e metodi, e molte imprese sono ancora legate alla vecchia impostazione artigianale. Però la maggior parte dei produttori di software ha adottato processi di sviluppo strutturati e metodologie di carattere ingegneristico.

1.2 Il software è diverso

Per comprendere meglio la storia ed i problemi della produzione del software, è bene aver presenti alcuni fatti che rendono il software un prodotto abbastanza diverso dagli altri manufatti:

- Prima di tutto, il software è "soft": non occupa praticamente spazio fisico, è invisibile, e gli aspetti materiali della sua produzione incidono pochissimo sul costo globale del prodotto; il software è informazione pura, e la sua produzione è progetto puro.
- Il software è estremamente *malleabile*, cioè modificabile con pochissimo sforzo. Questa proprietà è un vantaggio ma anche un pericolo, poiché incoraggia un modo di lavorare non pianificato.

- L'immaterialità del software fa sí che sia difficile visualizzare i concetti che gli sono propri, come algoritmi, strutture dati, e flussi di controllo. In effetti, una buona parte dell'ingegneria del software è rivolta allo studio e all'applicazione di notazioni espressive e sintetiche per rappresentare tali concetti.
- All'immaterialità del software è legata la difficoltà di quantificare e definire rigorosamente le caratteristiche del prodotto che ne determinano la qualità: mentre, per esempio, la qualità di una parte meccanica è largamente riconducibile a proprietà fisiche e quindi misurabili, questo generalmente non avviene per il software.
- Il software è *complesso*: un programma può essere composto da un numero grandissimo di istruzioni, e ciascuna di esse potenzialmente condiziona il comportamento di tutte le altre. Inoltre, il software è "*non lineare*", nel senso che un piccolo cambiamento nel codice può portare a grandi cambiamenti (spesso indesiderati!) nel funzionamento di un'applicazione.

A proposito della complessità del software, osserviamo che questa è tanto maggiore quanto meno il software è *strutturato*. Diciamo che un sistema software è strutturato se è suddiviso in componenti (*moduli*), ciascuno dei quali interagisce con pochi altri componenti. Le istruzioni contenute in ciascun componente hanno un campo d'azione limitato al componente stesso, eccettuate quelle istruzioni che devono espressamente interagire con altri componenti. Questo argomento verrà ripreso nel capitolo relativo al progetto del software.

Oltre alle caratteristiche del software sú esposte, sulla produzione del software pesa il fatto che si tratta di un'attività ancora giovane, sviluppatasi per di piú ad un ritmo velocissimo. Questo comporta varie conseguenze, fra cui la principale è forse una certa difficoltà di comprensione fra progettista e committente/utente: spesso un'organizzazione (o un individuo) sente il bisogno di uno strumento informatico per risolvere un problema, ma non ha un'idea chiara di cosa si possa chiedere allo strumento. Dall'altra parte, il progettista può non avere un'idea chiara delle esigenze dell'utente. Inoltre, in certi casi può anche mancare un linguaggio comune e consolidato sia fra progettista e utente che fra progettisti provenienti da ambienti diversi.

Si può anche osservare che, almeno fino a tempi recenti, l'impiego di strumenti automatici nello sviluppo del software è stato relativamente limitato, cosa abbastanza paradossale. Questo problema è collegato anche ad altre proprietà del software, come la sua complessità ed il carattere intel-

lettuale (quindi difficilmente automatizzabile) delle attività legate alla sua produzione.

Sebbene tutti questi problemi siano ancora sentiti, è però incoraggiante notare una generale maturazione nel mondo dell'industria informatica, per cui le conoscenze relative all'ingegneria del software sono sempre piú richieste ed apprezzate.

1.3 Il software non è diverso

Pur con le sue particolarità, il software è un prodotto industriale che come tutti i prodotti industriali deve soddisfare le esigenze dell'utente, quelle del produttore[1], e quelle della società. Quindi il prodotto deve svolgere le funzioni richieste, e deve farlo secondo i criteri di qualità richiesti. Deve anche essere realizzato e consegnato nei limiti di tempo richiesti (dettati sia dalle esigenze dell'utente che da fattori economici piú generali), e nei limiti di costo richiesti.

La possibilità di rispettare i vincoli sú esposti, e in generale di produrre il software in modo economicamente vantaggioso, dipende naturalmente dal *processo* di produzione, cioè dall'insieme delle attività volte alla produzione. Sebbene buona parte dell'ingegneria del software sia rivolta al prodotto, cioè alle teorie e alle tecniche necessarie per descriverne i requisiti, per realizzarlo e per valutarne la qualità, il suo obiettivo principale è lo studio del processo di produzione. Per questo si parla, come della definizione di Fairley, di una disciplina tecnologica e manageriale, che attinge da campi diversi, come la matematica, i vari settori dell'informatica, l'organizzazione aziendale, la psicologia, l'economia ed altri ancora.

L'insieme degli strumenti a disposizione dell'ingegnere del software è del tutto analogo a quello di cui si servono le altre discipline ingegneristiche:

Teorie, le basi formali e rigorose del lavoro. Per esempio, l'elettromagnetismo per l'ingegneria elettrica, la logica nell'ingegneria del software.

Linguaggi, notazioni (testuali o grafiche) e formalismi per esprimere i concetti usati e le loro realizzazioni: disegno tecnico per l'ingegneria meccanica, UML per il software.

[1] Non quelle del progettista, che comunque, nel caso dell'ingegnere informatico, si limitano alla disponibilità di un calcolatore, di lunghe notti al terminale, e di un po' di bevande e generi di conforto durante il lavoro `:)`.

Standard, documenti che stabiliscono requisiti uniformi che devono essere soddisfatti da processi, procedure, eccetera: norme UNI per l'ingegneria industriale, standard ANSI per l'informatica.

Strumenti, ambienti di sviluppo e programmi che assistono l'ingegnere nella progettazione, anche automatizzandone alcune fasi: programmi di calcolo delle strutture per l'ingegneria civile, strumenti CASE per il software.

Naturalmente, questo armamentario non è completo senza la capacità, da parte del progettista, di supplire con l'esperienza e l'inventiva ai limiti delle teorie e delle procedure formalizzate: è fondamentale conoscere le regole del mestiere, ma queste da sole non bastano per trovare tutte le soluzioni.

1.3.1 Deontologia

Si è accennato al fatto che ogni prodotto deve rispondere alle esigenze della società, oltre che a quelle del produttore e del cliente. Le esigenze della società sono in primo luogo quelle espresse da leggi e normative di vario genere. Al di là degli adempimenti legali, ogni progettista (come ogni altro individuo) è responsabile delle conseguenze che le proprie scelte ed i propri comportamenti possono avere sulla società: possiamo pensare, per esempio, all'impatto di un prodotto sull'ambiente, sui rapporti sociali, o sull'occupazione.

È particolarmente importante tener conto dei rischi economici e umani connessi all'uso del software. Il software è estremamente pervasivo, essendo un componente di sistemi disparati, come, per esempio, sistemi di comunicazione, elettrodomestici, veicoli, impianti industriali. Inoltre, alcuni prodotti software possono essere riusati in applicazioni diverse da quelle per cui sono stati concepiti originariamente, per cui il progettista di software può non essere in grado di anticipare la destinazione finale del proprio lavoro. Se sottovalutiamo la pervasività e l'adattabilità del software rischiamo di non valutare i rischi in modo adeguato. Le situazioni catastrofiche portate spesso ad esempio dei rischi del software (distruzione di veicoli spaziali, guasti in apparati militari o in centrali nucleari) possono non mettere in evidenza il fatto che del software difettoso si può trovare in qualsiasi officina o in qualsiasi automobile, mettendo a rischio la salute e la vita umana anche nelle situazioni piú comuni e quotidiane. Un ingegnere deve sentirsi sempre responsabile per il proprio lavoro, ed eseguirlo col massimo scrupolo, qualunque sia il suo campo di applicazione.

1.4 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 progettazione 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.

1.4.1 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 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 daagli utenti, ma spesso useremo il termine "utenti" per rife-

rirsi sia agli utenti propriamente detti che al committente, ove non sia necessario fare questa distinzione.

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*).

1.4.2 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 ne 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.

Come si è già detto, una specifica dice cosa fa un sistema, elencandone i requisiti, e l'implementazione dice come. 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/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. Se ciò non avviene, si potrebbe verificare una situazione di questo tipo: un sistema viene realizzato rispettando vincoli e requisiti, e dopo qualche tempo se ne fa una nuova versione. Nello sviluppare questa versione, una soluzione implementativa imposta dai requisiti o dai vincoli viene scambiata per una scelta di progetto, e viene sostituita da una versione alternativa, che può aggiungere qualcosa al sistema originale, però non rispetta i requisiti e i vincoli,

risultando quindi insufficiente. Supponiamo, per esempio, che i documenti di specifica chiedano che i record di un database possano essere elencati in ordine alfabetico, e che durante lo sviluppo dell'applicazione venga comunicato, *senza aggiornare la documentazione*, che deve essere rispettato un limite sull'uso di memoria centrale. Conseguentemente, sceglieremo un algoritmo di ordinamento efficiente in termini di memoria, anche se piú lento di altri. Se una versione successiva dell'applicazione viene sviluppata da persone ignare del vincolo sulla memoria, queste potrebbero sostituire l'algoritmo con uno piú veloce ma richiedente piú memoria, ed in questo modo verrebbe violato il vincolo, con la possibilità di malfunzionamenti.

1.4.3 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 seguito 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, oppure Sez. 1.1–1.2 Ghezzi, Fuggetta et al., oppure Cap. 1 Pressman.

Capitolo 2

Ciclo di vita e modelli di processo

Ogni prodotto industriale ha un *ciclo di vita* 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.

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 [3], MIL-STD-498 [22]), altri orientati a particolari settori di applicazione (per esempio, ECSS-E-40B [1] per l'industria spaziale, NS-G-1.1 [24] per l'industria nucleare, CEI EN 50128:2002-04 [7] per le ferrovie), ed altri ancora di applicabilità generale anche se prodotti per settori specifici (per esempio,

ESA PSS-05 [4], orientato all'industria spaziale).

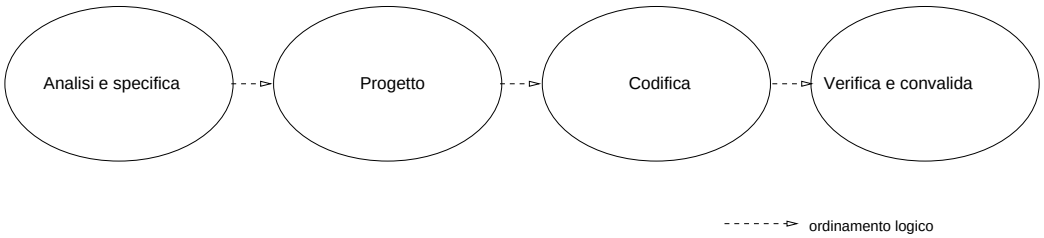

Figura 2.1: Il ciclo di vita del software.

Il ciclo di vita del software, di cui la Fig. 2.1 mostra le quattro attività relative allo sviluppo del software in senso stretto (tralasciando le attività di consegna, manutenzione e ritiro) segue lo schema generale appena esposto, ma con alcune importanti differenze, particolarmente nella fase di produzione. Come abbiamo visto, 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. 2.2). Successivamente si studieranno altri modelli di processo, in cui l'associazione fra attività e fasi del processo avviene in altri modi.

2.1 Il modello a cascata

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

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

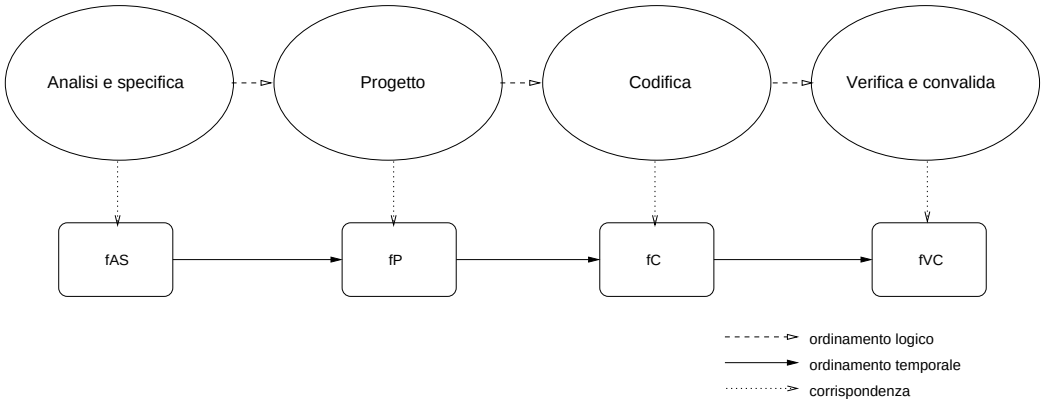

Figura 2.2: Il modello a cascata.

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.

Il numero, il contenuto e la denominazione delle fasi può variare da un'organizzazione all'altra, e da un progetto all'altro. Nel seguito 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.

2.1.1 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 valutare 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 cadere se il committente rinuncia, può essere affidato alla stessa organizzazione che ha fornito lo studio di fattibilità, oppure essere affidato ad un'altra organizzazione. In questo caso 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.

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.

2.1.2 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 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 [26] 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 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, le descrizioni 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 [26]). 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.

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 seguito.

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 (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.

È di fondamentale importanza che il DSR sia *completo* e *consistente*. "Completo" significa che contiene esplicitamente tutte le informazioni necessarie, e "consistente" significa che non contiene requisiti reciprocamente contraddittori. Idealmente, dovrebbe essere scritto in un linguaggio formale, tale da permettere un'interpretazione non ambigua ed una verifica rigorosa, ma di solito è in linguaggio naturale, al piú strutturato secondo qualche standard, e accompagnato da notazioni semiformali (diagrammi etc.).

A proposito dell'uso del linguaggio naturale nei documenti di specifica, osserviamo che il significato dei termini usati può essere definito dai *glossari*. Per ogni progetto è opportuno preparare un glossario dei termini ad esso specifici; inoltre, esistono numerosi glossari standard, come, per esempio, la norma ISO-8402 relativa alla terminologia della gestione e assicurazione della qualità. Un documento può anche contenere istruzioni relative ad usi particolari (generalmente piú ristretti) di termini appartenenti al linguaggio ordinario. Per esempio, è uso comune che il verbo (inglese) "shall" denoti comportamenti o caratteristiche obbligatori, il verbo "should" denoti comportamenti o caratteristiche desiderabili (ma non obbligatori), ed il verbo "may" denoti comportamenti o caratteristiche permessi o possibili.

Il DSR deve riportare almeno queste informazioni [14]:

- 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.

2.1.3 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 principi e criteri che verrano 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. 6.5.1).

2.1.4 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.

Evidentemente questa fase è cruciale in quanto consiste nella realizzazione pratica di tutte le specifiche e le scelte delle fasi precedenti. Il codice prodotto in questa fase, oltre ad essere conforme al progetto, deve avere delle qualità che, pur essendo invisibili all'utente, concorrono in modo determinante sia alla bontà del prodotto che all'efficacia del processo di produzione. Fra queste qualità citiamo la *modificabilità* e la *leggibilità*. Per conseguire tali qualità è necessario adottare degli standard di codifica, che stabiliscono, ad esempio, il formato (nel senso tipografico) dei file sorgente, le informazioni che devono contenere oltre al codice (autore, identificazione del modulo e della versione...), ed altre convenzioni.

L'attività di codifica è strettamente collegata a quella di testing e di debugging. Tradizionalmente queste tre attività sono affidate alla stessa persona, che le esegue secondo i propri criteri. Tuttavia l'attività di testing di unità, per la sua influenza critica sulla qualità del prodotto, deve essere svolta in modo metodico e pianificato, e si deve avvalere di metodologie specifiche, come previsto, p. es., dallo standard ANSI/IEEE 1008-1987 *Standard for Software Unit Testing* [2].

In questo corso non si parlerà della programmazione, di cui si suppongono noti i principî 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. Un'altro strumento di questo tipo è il *Concurrent Versioning System* (*CVS*)[3].

Configurazione e compilazione automatica. Esistono strumenti che permettono di automatizzare il processo di costruzione (compilazione e collegamento) del software, e di configurare questo processo, cioè di adattarlo a diverse piattaforme software. Alcuni di questi strumenti sono i programmi *Make*, *Automake*, *Autoconf*, e *Libtool*, noti collettivamente come *Autotools* [4].

Notifica e archiviazione di malfunzionamenti. Lo strumento *Bugzilla*, basato su web, permette 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*[5].

Test di unità. Il test di unità può essere parzialmente automatizzato grazie a strumenti come *DejaGNU*[6], *CppUnit*[7], *mockpp*[8].

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.

2.1.5 Integrazione e test di sistema

In questa fase viene assemblato 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.

[1] v. `subversion.tigris.org`
[2] Si pronuncia con l'accento tonico sulla seconda sillaba.
[3] v. `www.cvshome.org`
[4] v. `www.gnu.org/manual`
[5] v. `www.mozilla.org/projects/bugzilla`
[6] v. `www.gnu.org/software/dejagnu`
[7] v. `cppunit.sourceforge.net`
[8] v. `mockpp.sourceforge.net`

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

Il test di sistema può essere seguito, quando l'applicazione è indirizzata al mercato, da questi test:

alfa test: l'applicazione viene usata all'interno all'azienda produttrice;
beta test: l'applicazione viene usata da pochi utenti esterni selezionati (*beta tester*).

2.1.6 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 riparazione di difetti presenti nel prodotto consegnato al committente, oppure l'aggiornamento del codice allo scopo di fornire nuove versioni. Questa riparazione consiste nel modificare e ricostruire il software. Si dovrebbe quindi parlare di *riprogettazione* piuttosto che di manutenzione. La scoperta di difetti 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*) 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 principio è chiamato "progettare per il cambiamento" (*design for change*). 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);

perfettiva: aggiunte e miglioramenti.

2.1.7 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, verbali 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*.

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 dei semilavorati ottenuti dalle fasi intermedie.

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 si aspettano dal prodotto software, e sono espressi in modo informale ed approssimativo, quando non restano impliciti e sottintesi; le specifiche sono l'espressione esplicita e rigorosa dei requisiti (Fig. 2.3).

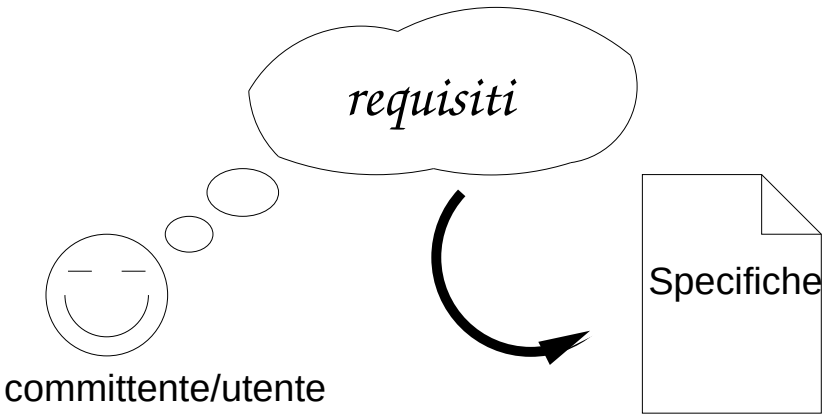

Figura 2.3: Requisiti e specifiche.

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. 2.4, da [15]).

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 la soltanto la valutazione del prodotto *finale* rispetto ai requisiti, mentre la verifica si applica ai semilavorati di ciascuna fase intermedia.

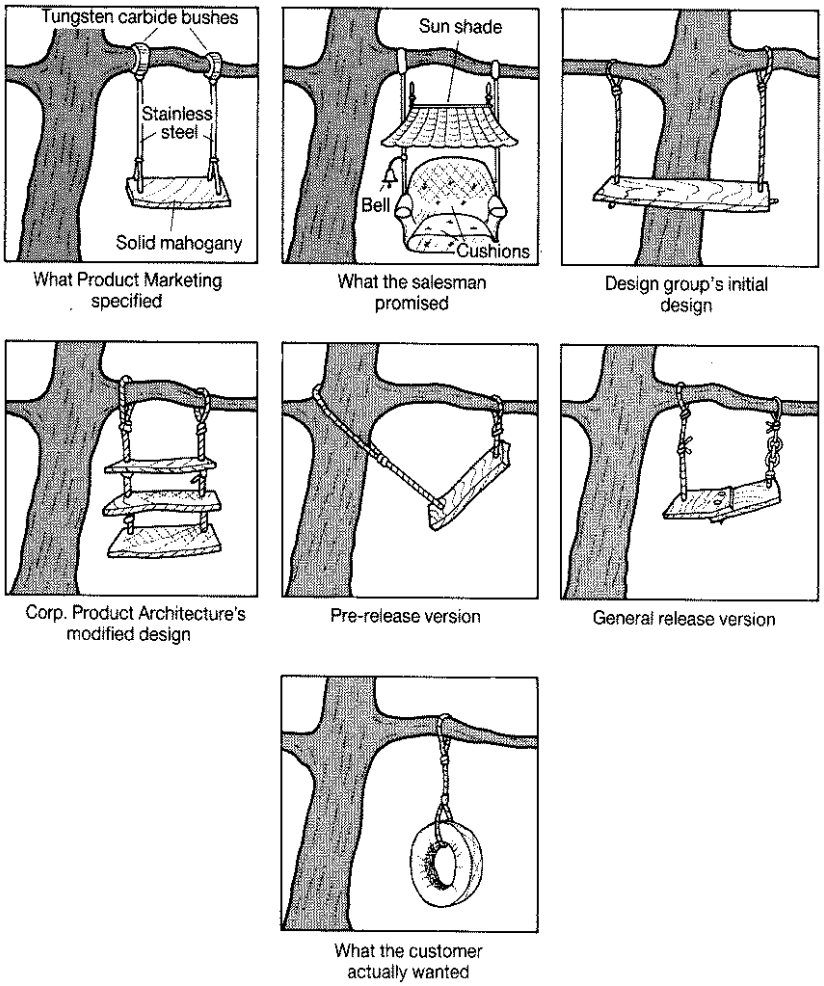

Figura 2.4: Il processo di sviluppo del software [15].

2.2 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. 2.5 mostra un esempio di processo a V, dallo standard IAEA TRS 384 [6].

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*).

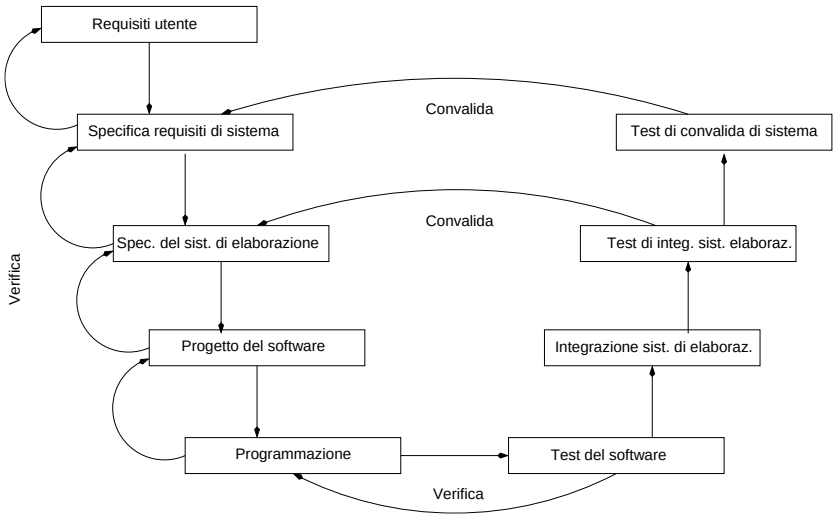

Figura 2.5: Un processo a V (da IAEA TRS 384, modificato).

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.

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

Un altro esempio [27] di processo a V è mostrato in Fig. 2.6. Questo processo descrive specificamente lo sviluppo del sistema software.

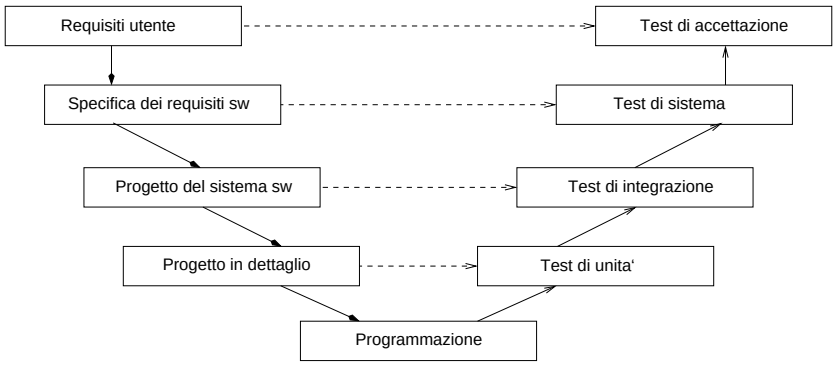

Figura 2.6: Un processo a V (da [27]).

2.3 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.

Nei processi basati su modelli evolutivi, il software viene prodotto in modo incrementale, in passi successivi (Fig. 2.7). 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 risponde pienamente alle esigenze del committente, almeno finché non si sentirà il bisogno di una nuova versione.

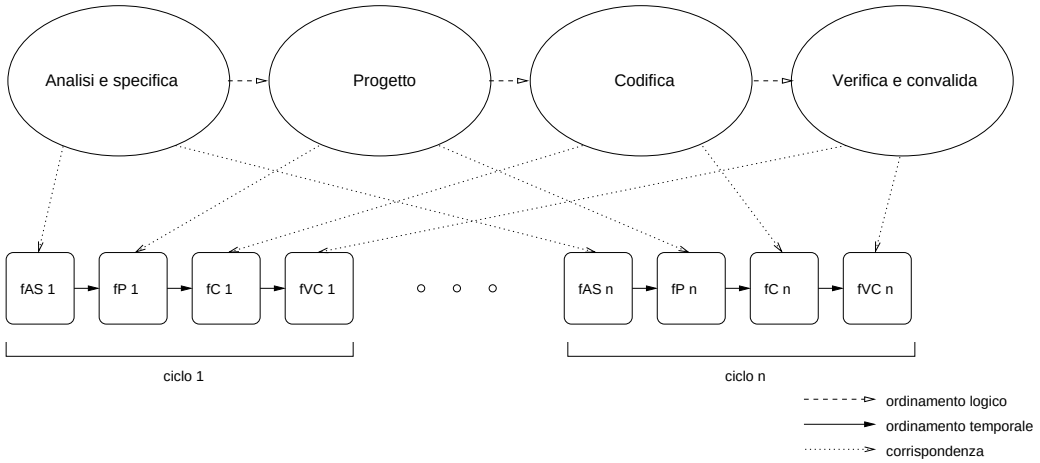

Figura 2.7: Un processo incrementale.

2.3.1 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 modelli evolutivi in diversi modi [19]:

- 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 può essere 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*).

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 alla 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) o linguaggi di scripting (p.es., Perl 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.

2.3.2 Lo Unified Process

Lo Unified Process (UP) [16, 9] è un processo evolutivo concepito per lo sviluppo di software orientato agli oggetti ed è stato 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* (*requirements*), *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*.

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 esere 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.

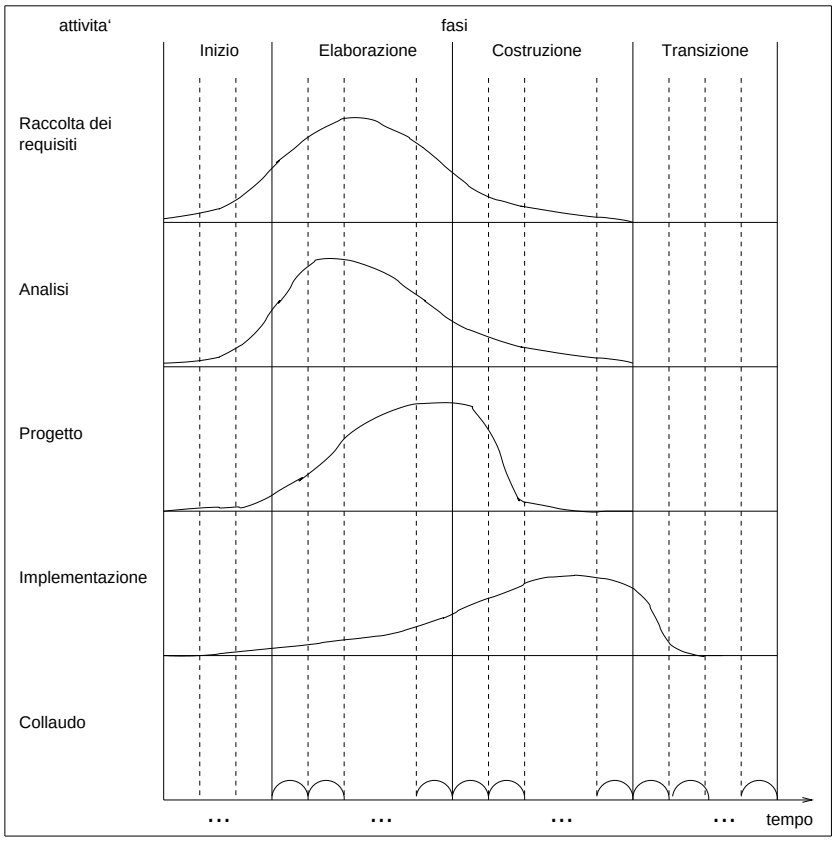

Figura 2.8: Fasi e attività nello Unified Process.

Questo andamento è schematizzato in Fig. 2.8. 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.

2.3.3 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:

$$f(x, y) = g(x, y)$$

$$\ldots$$

$$y = h(x)$$

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

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 b. La Fig. 2.9 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, 8]. In questo linguaggio sono definiti numerosi operatori

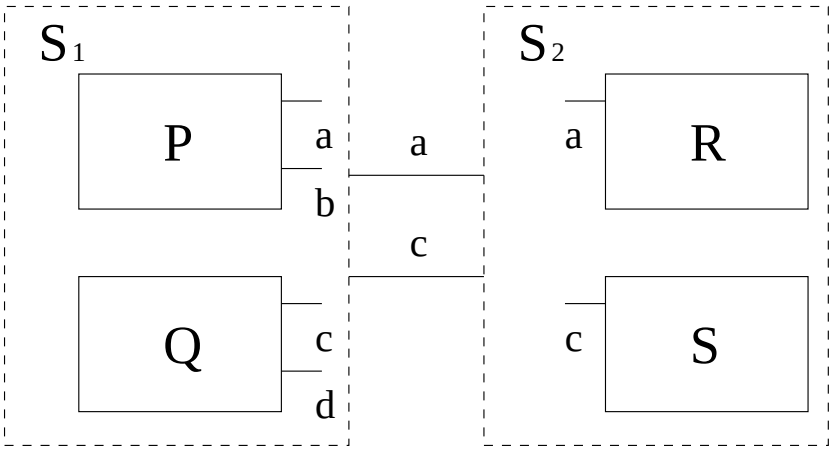

Figura 2.9: Struttura risultante dalla specifica iniziale.

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.

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. 2.10). Il nuovo raggruppamento dei processi è presumibilmente migliore del precedente, perché mette insieme i processi che devono interagire e separa quelli reciprocamente indipendenti.

Processo di sviluppo

Un processo trasformazionale generalmente è assistito da uno strumento CASE che esegue le trasformazioni scelte dal progettista e, qualora vengano introdotte nuove informazioni (per esempio, requisiti o vincoli aggiuntivi), ne verifica la consistenza rispetto a quelle già esistenti.

Lo strumente CASE può permettere di associare ad ogni scelta le relative motivazioni. Viene cosí costruito un database di progetto (*repository*) contenente tutte le informazioni relative alla storia ed allo stato attuale del progetto. Osserviamo però che l'uso del repository non è limitato ai processi di tipo trasformazionale.

L'ambiente di progetto può comprendere un sistema di gestione delle configurazioni che impedisce di modificare il codice senza prima modificare

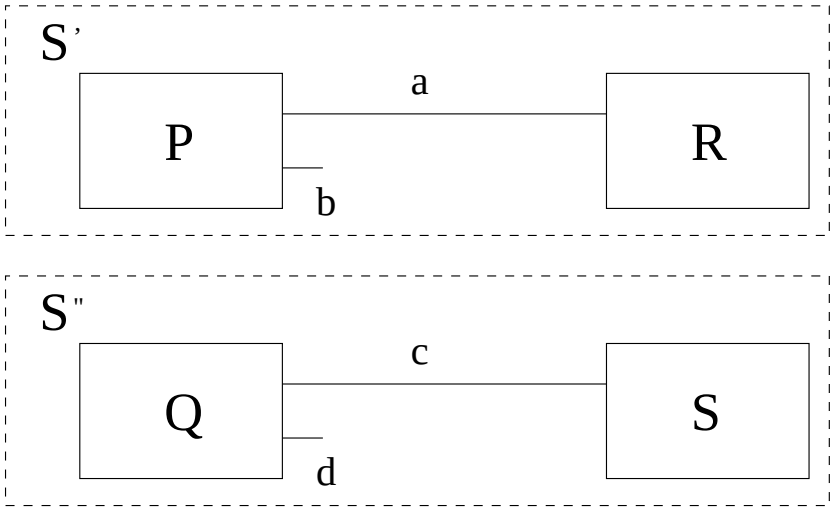

Figura 2.10: Struttura trasformata.

le specifiche rilevanti. Dopo che sono state aggiornate le specifiche al livello di astrazione appropriato, viene ricostruito il processo di derivazione fino ad ottenere una nuova versione del software. Si evita cosí quella degradazione del software a cui si è accennato parlando della manutenzione.

Se le specifiche sono espresse in un linguaggio eseguibile, come avviene di norma, allora ciascuna specifica è un prototipo di tipo evolutivo. Il modello trasformazionale è quindi caratterizzato dalla prototipazione e dall'uso di linguaggi formali. È concepibile un processo di tipo trasformazionale che non usi linguaggi di specifica eseguibili, ma generalmente sono disponibili degli interpreti per i linguaggi formali di specifica, che rendono possibile la prototipazione.

Il prodotto finale può essere implementato nello stesso linguaggio delle specifiche, o in un linguaggio diverso.

Letture

Obbligatorie: Cap. 7 Ghezzi, Jazayeri, Mandrioli, oppure Cap. 1 Ghezzi, Fuggetta et al., oppure Cap. 3 Pressman.

Facoltative: Sez. 13.7.2 Pressman, Cap. 2 Arlow, Neustadt.