Marco Cadoli, Maurizio Lenzerini,
Paolo Naggar, Andrea Schaerf

Fondamenti della progettazione dei programmi

Princìpi, tecniche e loro applicazioni in C++

CittàStudiEdizioni

Prefazione

Questo testo presenta i fondamenti della progettazione dei programmi, illustrando i princìpi, le tecniche, gli strumenti della progettazione e la loro applicazione nello sviluppo di programmi nel linguaggio C++. Il testo nasce dall'esperienza maturata dagli autori in diversi anni di insegnamento di corsi universitari e professionali riguardanti la progettazione dei programmi, ed è rivolto sia agli studenti universitari sia ai professionisti che vogliano approfondire gli aspetti metodologici della progettazione del software.

Obiettivi

L'obiettivo del testo è di presentare i princìpi, le tecniche e le metodologie per la progettazione e la realizzazione di programmi, mantenendo un giusto equilibrio fra aspetti formali, concettuali e metodologici e aspetti legati alla pratica della programmazione in C++. Il linguaggio C++ viene utilizzato come veicolo per la realizzazione pratica delle idee presentate. L'enfasi non viene posta quindi sulle caratteristiche del linguaggio in quanto tali, ma su concetti, tecniche e metodi di carattere generale, e sul modo in cui queste nozioni possono trovare applicazione attraverso tali caratteristiche. Il testo propone comunque una vasta gamma di realizzazioni di moduli software in C++, e fornisce un nutrito insieme di tecniche che indirizzano ad un uso corretto del linguaggio.

Prerequisiti

Si assume che il lettore abbia familiarità con le nozioni base della programmazione, e con un linguaggio di programmazione imperativa ad alto livello, come il PASCAL. Il livello di conoscenza dei suddetti argomenti che si richiede è, ad esempio, quello della trattazione in [3]. Il testo comprende alcuni capitoli introduttivi sul linguaggio C++, che sono sufficienti per la comprensione dei programmi riportati nei capitoli successivi, ma che non forniscono una descrizione completa del linguaggio. A coloro che vogliono approfondire le caratteristiche del C++ suggeriamo di affiancare lo studio di questo testo alla lettura di libri specifici come [33] e alla consultazione del manuale del compilatore utilizzato per le esercitazioni.

Contenuti

Il testo è organizzato in sei parti.

1. *La progettazione dei programmi.* Viene descritto sinteticamente il contesto organizzativo in cui si situa lo sviluppo del software, si illustrano le fasi principali in cui viene tradizionalmente suddiviso il processo di sviluppo e si discutono i

fattori mediante i quali si giudicano le qualità di un programma. Si approfondiscono poi le caratteristiche fondamentali della fase di progettazione, fornendo una panoramica sui princìpi, le tecniche e gli strumenti di progettazione.

2. *Introduzione al C++.* Vengono introdotte le nozioni fondamentali del linguaggio C++. Particolare attenzione viene rivolta alle nozioni di classe e di ereditarietà, che verranno ampiamente utilizzate nei successivi capitoli del libro.

3. *Basi matematiche.* Vengono illustrate le proprietà fondamentali di tre sistemi matematici che forniscono importanti nozioni di base per la progettazione dei programmi: la logica, l'algebra e la complessità computazionale.

4. *La modularizzazione.* Viene presentata la tecnica della modularizzazione, che consente di strutturare un programma in parti indipendenti e interagenti, e che rappresenta una delle più importanti nozioni di base della progettazione dei programmi. Vengono illustrati i criteri per produrre programmi modulari, vengono descritti diversi tipi di modularizzazione, e viene presentato uno schema di metodologia di progettazione basato su questa tecnica. Successivamente, l'attenzione si concentra sulla modularizzazione per tipo astratto, secondo la quale vengono sviluppati moduli che realizzano tipi astratti di dato significativi per la applicazione, e che è la base delle metodologie di sviluppo e di progettazione orientate agli oggetti.

5. *Le tecniche algoritmiche.* Vengono illustrati gli strumenti concettuali e metodologici che portano dall'identificazione di un problema alla realizzazione di un algoritmo e di un programma in grado di risolvere tale problema. Viene presentato uno schema di riferimento che permette di esprimere in modo preciso le caratteristiche dei problemi, e vengono illustrate quattro importanti tecniche per la progettazione di algoritmi. Ogni tecnica è descritta in modo generale, applicata ad alcuni esempi, e discussa sia dal punto di vista della sua applicabilità, sia dal punto di vista della correttezza e dell'efficienza dei corrispondenti algoritmi.

6. *Esempi di progettazione.* Vengono presentati tre esempi di progettazione, evidenziando l'applicazione dei princìpi e delle tecniche illustrate nei precedenti capitoli.

Ciascun capitolo di ogni parte è corredato di esempi ed esercizi, e si conclude con un paragrafo sulle note bibliografiche relative agli argomenti trattati, ed uno in cui si propongono ulteriori esercizi di varia difficoltà. In appendice viene presentata la soluzione di alcuni degli esercizi. Il testo si conclude con la lista dei riferimenti bibliografici.

Utilizzo didattico

Questo testo nasce dall'esperienza didattica degli autori, che hanno insegnato per molti anni nell'ambito del corso di Fondamenti di Informatica II presso la Facoltà di Ingegneria dell'Università di Roma "La Sapienza", e che hanno tenuto numerosi corsi professionali in altri contesti.

Il testo può essere adottato come materiale didattico per un tipico insegnamento annuale nell'ambito di un corso di Laurea, il cui carico didattico è di circa settantacinque ore di lezione e trentacinque ore di esercitazione. Insegnamenti che possono

utilizzare il testo con questa modalità vengono erogati nei corsi di Laurea delle Facoltà di Ingegneria, di Scienze dell'Informazione e di Informatica.

Il testo può anche essere utilizzato in due mezze annualità nell'ambito dei Corsi di Diploma Universitario delle suddette Facoltà. In questo caso, ogni mezza annualità corrisponde ad un modulo didattico di circa trenta ore di lezione e venti di esercitazione. Si suggerisce di utilizzare le parti I, II, IV (tralasciando i capitoli 11, 13 e 14) per un modulo, e le parti IV (capitoli 13 e 14), V e VI per un altro modulo.

Infine, il testo può essere adottato in corsi professionali secondo diversi percorsi didattici.

Note sui programmi

Tutti i programmi contenuti in questo testo sono stati compilati, eseguiti e verificati usando il compilatore `CC` (versione 4.1) della Sun Microsystems su macchina Sun `SPARCstation10` con sistema operativo UNIX Solaris (versione 2.5). Nella stesura dei programmi sono state utilizzate solamente caratteristiche standard del linguaggio C++. Nonostante ciò, a causa delle differenze fra i vari compilatori esistenti, alcuni programmi, in particolare quelli che fanno uso di `template`, potrebbero non essere compilabili in altri ambienti.

Il codice dei programmi è disponibile su Internet, e gli interessati possono ottenerlo in due modi:

- accedendo alla pagina WWW all'indirizzo

  `http://www.dis.uniroma1.it/pub/clns`

- per mezzo di `ftp` anonimo all'indirizzo

  `dis.uniroma1.it`

  nel direttorio `pub/clns`

I file in cui si trovano i programmi sono organizzati in direttori sulla base dei capitoli in cui sono presentati. Ad esempio, il direttorio che raccoglie i file con i programmi del capitolo 16 è identificato dal nome `Cap16`. Anche nel testo ci riferiremo a questa organizzazione dei file. Così, nei programmi all'interno di un capitolo, per riferirsi ad un file contenuto in un direttorio corrispondente ad un altro capitolo, useremo la notazione del sistema UNIX. Ad esempio, per includere il file `tempo.h` del capitolo 4 in un programma presentato in un altro capitolo scriveremo la direttiva `#include "../Cap04/tempo.h"`.

Ringraziamenti

Gli autori desiderano ringraziare Daniele Nardi e Giuseppe De Giacomo per utili commenti e consigli, e gli studenti dei corsi di Metodologie di Programmazione e Fondamenti di Informatica II della Facoltà di Ingegneria dell'Università di Roma "La Sapienza", che hanno utilizzato in diversi anni accademici, sotto forma di dispense, parte del materiale presente in questo testo.

Un ringraziamento speciale va al Dipartimento di Informatica e Sistemistica dell'Università di Roma "La Sapienza", per aver messo a disposizione gli strumenti di calcolo utilizzati sia per la compilazione e l'esecuzione dei programmi, sia per la diffusione di questi ultimi tramite Internet.

Parte I

La progettazione dei programmi

Lo scopo di questa parte è di fornire un quadro generale sulla progettazione dei programmi.

Nel capitolo 1 descriviamo sinteticamente il contesto organizzativo in cui si situa lo sviluppo del software, illustriamo le fasi principali in cui viene tradizionalmente suddiviso il processo di sviluppo, e discutiamo un insieme di fattori mediante i quali si giudica la qualità di un programma.

Il capitolo 2 è dedicato alla illustrazione delle nozioni di base relative alla fase di progettazione, che è il principale oggetto di indagine di questo testo. Presenteremo queste nozioni distinguendo tra princìpi, tecniche, e strumenti di progettazione. I princìpi di progettazione sono le linee guida generali che dovrebbero essere alla base di ogni scelta e strategia di progetto. Le tecniche sono i metodi che si adottano al fine di condurre il progetto in modo coerente con i princìpi. Infine, gli strumenti sono formalismi, sistemi, e ausili di varia natura che supportano le attività di progettazione e guidano il progettista nell'applicazione delle tecniche.

Capitolo 1

Considerazioni generali sullo sviluppo di programmi

In questo capitolo illustriamo il quadro metodologico generale in cui viene condotto lo sviluppo dei programmi, al fine di collocare in un appropriato contesto le considerazioni relative alla fase di progetto, che costituisce il principale oggetto di indagine del presente testo.

Forniamo dapprima una breve descrizione del contesto organizzativo in cui viene in genere sviluppato il software. Illustriamo successivamente le diverse fasi in cui viene tradizionalmente scomposto il processo di sviluppo del software. Infine, presentiamo un insieme di fattori di qualità che consentono di giudicare il prodotto software rispetto a diversi punti di vista.

1.1 Il contesto organizzativo

I programmi vengono progettati e redatti a seguito di una esigenza espressa da un *committente*, cioè un individuo o una organizzazione che necessita di una soluzione informatica ad un dato problema o una data applicazione. Il committente incarica quindi, in genere sulla base di un contratto, un *progettista* o un gruppo di progettisti di sviluppare il programma. Usiamo il termine generico di progettista per indicare diverse figure che prendono parte all'attività di progetto. Alcune volte si distingue tra *analista* e *programmatore* per riferirsi ai diversi ruoli che i progettisti svolgono durante lo sviluppo del programma. L'analista conduce attività rivolte alla definizione di cosa deve fare il programma, mentre il programmatore svolge l'attività di codifica del programma stesso.

Una volta prodotto, il programma va in esercizio, e in questa fase viene utilizzato direttamente dal cosiddetto *utente finale*. L'utente finale può essere il committente stesso, oppure un qualunque individuo, organizzazione o sistema che è interessato direttamente all'esecuzione del programma e all'utilizzo dei risultati prodotti. Il programma in esercizio ha spesso bisogno di modifiche, aggiunte, e/o correzioni. Queste vengono svolte dal cosiddetto *manutentore*, che può o no coincidere con il progettista che ha sviluppato il programma.

Da queste sintetiche osservazioni deriva che, in questa sede, per programma intendiamo un prodotto software, di qualunque dimensione, che viene sviluppato allo scopo di svolgere determinate funzioni elaborative. In particolare, nella nostra accezione, un programma può essere sia un prodotto di piccole dimensioni, sviluppato da un unico progettista a fronte di una ben specificata esigenza elaborativa, sia un prodotto più complesso, composto da diversi componenti, eventualmente realizzati da diversi progettisti e programmatori, che concorrono tutti alla realizzazione della soluzione informatica di uno specifico problema elaborativo. È evidente che le metodologie di progetto dei programmi del secondo tipo saranno più complesse e articolate, e terranno anche conto di problemi di coordinamento e comunicazione tra i diversi componenti del gruppo di progetto, ma è altrettanto chiaro che le considerazioni generali sul processo di sviluppo prescindono dai suddetti fattori, e possono essere svolte ad un livello sufficientemente generale da fornire un quadro unitario sul prodotto programma in quanto tale.

Al quadro appena presentato dobbiamo anche aggiungere alcune considerazioni sulle classi di applicazioni per le quali il software viene realizzato. Per fornire alcune idee di carattere generale, possiamo fare riferimento a due classificazioni riportate in [22].

Una prima classificazione riguarda le proprietà del sistema rispetto al flusso di esecuzione delle attività che lo compongono, e distingue tra:

- Applicazioni *sequenziali*, che sono caratterizzate da un unico flusso di controllo che governa l'evoluzione dell'applicazione. Queste sono le applicazioni più tradizionali, e vengono spesso adottate come riferimento per i metodi e le tecniche di base per la progettazione.

- Applicazioni *concorrenti*, che sono caratterizzate dal fatto che le varie attività che compongono il sistema non hanno una natura inerentemente sequenziale, ma necessitano di meccanismi di sincronizzazione e comunicazione.

- Applicazioni *dipendenti dal tempo*, che sono influenzate da vincoli temporali riguardanti sia la velocità di esecuzione delle attività sia la necessità di sincronizzare le attività stesse.

Una seconda classificazione riguarda invece la natura degli elementi che sono di interesse primario nella applicazione, e distingue tra:

- Applicazioni *orientate alla realizzazione di funzioni*, in cui la complessità prevalente del sistema riguarda le funzioni da realizzare.

- Applicazioni *orientate alla gestione di dati*, in cui l'aspetto prevalente è rappresentato dai dati che vengono memorizzati, ricercati, e modificati, e che costituiscono il patrimonio informativo di una organizzazione. Si usa spesso il nome di *sistema informativo* per indicare un sistema software realizzato per questo tipo di applicazioni.

- Applicazioni *orientate al controllo*, in cui la complessità prevalente del sistema riguarda il controllo delle attività che si sincronizzano e cooperano durante l'evoluzione del sistema.

In questo testo siamo interessati ai fondamenti della progettazione, e rivolgiamo quindi la nostra attenzione a metodi e tecniche che rappresentano la base per ogni

classe di applicazioni. Faremo perciò riferimento alla classe di applicazioni più semplici, e cioè quelle sequenziali orientate alla realizzazione di funzioni. Metodologie specifiche per le altre classi di applicazioni sono al di fuori degli obiettivi di questo testo, e si possono considerare come estensioni ai metodi e alle tecniche qui descritte.

1.2 Il ciclo di sviluppo dei programmi

L'obiettivo di questo paragrafo è l'illustrazione delle diverse fasi che compongono il processo di sviluppo dei programmi. La trattazione ha lo scopo di inquadrare la fase di progettazione, e viene quindi condotta a livello generale, senza pretese di completezza.

Il ciclo di sviluppo di un programma viene tradizionalmente visto come composto dalle seguenti fasi:

1. Studio di fattibilità
2. Raccolta e analisi dei requisiti
3. Progettazione
4. Verifica
5. Manutenzione

La fase di *studio di fattibilità* ha lo scopo di effettuare una valutazione dei costi e dei benefici del sistema che si intende costruire, con l'obiettivo di stabilire se e quando il progetto deve iniziare, di analizzare quali siano le alternative possibili per lo sviluppo e la realizzazione del sistema, e di individuare le risorse necessarie per l'attuazione del progetto.

La fase di *raccolta e analisi dei requisiti* ha la scopo di definire con precisione il problema da risolvere e, a fronte delle caratteristiche individuate, di specificare l'ambiente di sviluppo e l'ambiente di realizzazione, sia hardware (sistema di elaborazione, eventuale rete di comunicazione ecc.) che software (linguaggio di programmazione, ambiente di programmazione ecc.). Il risultato di questa fase è, in genere, un documento, detto *documento di analisi*, che specifica tutti gli aspetti sopra menzionati. Il livello di precisione e di formalità di tale documento varia da caso a caso: in un grosso progetto a cui partecipano diversi progettisti e programmatori può essere un documento formale, redatto secondo criteri prestabiliti, mentre in un progetto di piccole dimensioni può essere un testo che specifica informalmente il problema in esame.

Il documento di analisi si compone in genere di due parti:

- Una parte che specifica l'ambiente hardware e software scelto per l'applicazione, e descrive i requisiti raccolti, illustrando, spesso discorsivamente, le caratteristiche del problema elaborativo da risolvere.

- Una parte che specifica, mediante formalismi ad hoc, i diversi aspetti dell'applicazione da realizzare. In genere, gli aspetti che si prendono in considerazione sono:

  - La struttura concettuale dei dati che il sistema deve manipolare.
  - Le caratteristiche generali delle operazioni che il sistema deve effettuare, e le dipendenze che si creano tra esse in virtù dei flussi di informazione che si devono garantire nel sistema.

– L'evoluzione del sistema complessivo in termini di flusso di esecuzione delle varie operazioni al fine del raggiungimento degli obiettivi elaborativi del sistema.

Non è nostro scopo l'approfondimento dei metodi, dei formalismi, e degli strumenti che si utilizzano nella fase di raccolta e analisi dei requisiti. Vogliamo solo sottolineare che il documento di analisi è un prodotto importante nel ciclo di sviluppo, sia perché costituisce il punto di riferimento per tutte le fasi seguenti, sia perché rappresenta il principale input alla fase di progettazione.

La fase di *progettazione* ha lo scopo di concepire la soluzione informatica del problema, di definire l'architettura del programma, di individuare e specificare l'organizzazione del programma in parti ben definite ed interagenti, di specificare le funzioni delle componenti individuate, di scegliere le strutture di rappresentazione degli oggetti da manipolare, e di redigere il programma secondo il linguaggio e l'ambiente di programmazione scelto. Il risultato di questa fase è il programma vero e proprio, spesso corredato da altri documenti (integrati o meno nel testo del programma) che descrivono sia il processo che ha portato al programma, sia il programma stesso. I princìpi e le tecniche che sono alla base di questa fase costituiscono l'oggetto di indagine di questo testo.

La fase di *verifica* ha lo scopo di analizzare tutta la documentazione prodotta nelle fasi di analisi e di progettazione allo scopo di verificare che il programma prodotto svolga correttamente, completamente, ed efficientemente il compito per il quale è stato sviluppato. La verifica viene condotta sia mediante analisi del testo del programma, sia mediante una serie di prove su dati di test. A seconda del risultato di questa fase, sarà ovviamente possibile tornare a svolgere attività proprie della fase di progetto, ad esempio per modificare scelte dimostratesi erronee o non completamente adeguate, o per correggere possibili errori di varia natura.

La fase di *manutenzione* ha lo scopo da una parte di controllare che il programma, durante il periodo di esercizio, produca in ogni situazione i risultati attesi secondo le specifiche, e dall'altra di correggere eventuali errori e disfunzioni, e aggiornare il programma a fronte di eventuali cambiamenti nelle specifiche. Anche in questa fase sarà ovviamente possibile un ritorno ad attività della fase di progetto.

Il ciclo di sviluppo che abbiamo appena descritto viene spesso chiamato *a cascata*, per sottolineare che la struttura del processo si compone di fasi che, almeno a livello schematico, dovrebbero essere eseguite sequenzialmente. Osserviamo, però, che esso rappresenta più un modello di riferimento che una vera metodologia rigida e prescrittiva. Nella realtà, è infatti opportuno condurre il processo di sviluppo in modo flessibile, per garantire che il prodotto finale sia effettivamente costruito secondo le esigenze. A tale scopo è spesso opportuno prevedere che, durante la fase di progetto, o almeno subito dopo, si costruisca un prototipo del programma, cioè una versione che, pur non avendo tutte le caratteristiche di completezza della soluzione finale, evidenzi le scelte più critiche e prefiguri il comportamento del prodotto quando sarà nella fase di esercizio. È evidente che, una volta prodotto il prototipo, la fase di verifica potrà svolgersi più efficacemente interagendo con l'utente finale, che avrà a disposizione elementi concreti per giudicare la bontà delle scelte effettuate. Tenendo conto di questo ulteriore elemento, il processo di sviluppo si comporrà di una successione di rilasci di versioni sempre più accurate, complete, e rispondenti alle esigenze dell'utente. In questo modo, la suddivisione nelle fasi di progetto, verifica e manutenzione è meno netta di quella rappresentata dal modello a cascata. Per una descrizione più dettagliata di altri modelli di descrizione del ciclo di vita del software rimandiamo a [22].

1.3 Le qualità dei programmi

Qualunque trattazione sulla fase di progettazione non può prescindere da un'analisi delle qualità che i programmi devono possedere. Le qualità che in genere si attribuiscono ai programmi sono di due tipi: esterne ed interne. Le prime si riferiscono a caratteristiche evidenziabili dal solo funzionamento del programma durante la fase di esercizio e di interazione con l'utente. Tipico esempio di qualità di questa classe è la correttezza, cioè la capacità del programma di svolgere in modo soddisfacente i compiti per il quale è stato realizzato. Le seconde si riferiscono invece alle caratteristiche del programma che sono analizzabili e valutabili, tipicamente da esperti, solo mediante uno studio delle scelte tecniche adottate. Ad esempio, la leggibilità, intesa come la possibilità di comprendere sia la struttura del programma che le funzioni svolte dalle varie parti del programma, è una qualità interna.

- Le più importanti *qualità esterne* sono la correttezza, l'efficienza, la robustezza e l'usabilità:

  1. Come abbiamo già detto, *la correttezza* è la capacità del programma di eseguire precisamente i compiti individuati durante l'analisi dei requisiti. È evidente che la correttezza è la qualità primaria del software, anche se è forse la più difficile da valutare in modo oggettivo ed affidabile. Per la valutazione formale della correttezza è infatti necessario da una parte che i requisiti siano espressi in modo formale, e dall'altra che l'attività di verifica venga condotta utilizzando strumenti e formalismi matematici ad hoc. Sebbene dedicheremo nel seguito molta attenzione alla problematica della correttezza dei programmi, notiamo fin d'ora che la verifica formale della correttezza esula dagli scopi di questo testo.
  2. L'*efficienza* è la capacità del programma di utilizzare in modo razionale ed economico le risorse di calcolo. L'efficienza viene tipicamente misurata rispetto a due risorse fondamentali, il tempo di esecuzione e la quantità di memoria. Anche per la valutazione dell'efficienza vengono impiegate tecniche e strumenti matematici ad hoc. Tra questi, faremo spesso riferimento alla complessità asintotica di tempo e spazio, per la quale utilizzeremo la usuale notazione O (cfr. capitolo 8).
  3. La *robustezza* è la capacità del programma di funzionare in modo soddisfacente in condizioni anomale rispetto a quelle previste in fase di analisi dei requisiti. Mentre la correttezza riguarda le condizioni normali di funzionamento, la robustezza è relativa al funzionamento del programma nei casi in cui le condizioni esterne (dati di ingresso, interfacce ecc.) rivelino caratteristiche impreviste. Si noti che situazioni anomale si presentano molto più spesso di quanto si potrebbe pensare, specialmente in considerazione del fatto che è praticamente impossibile che nell'analisi dei requisiti siano stati considerati tutti i casi che si possono verificare nel funzionamento del programma. È quindi importante che il programma reagisca a queste situazioni in modo controllato, proteggendo, ad esempio, informazioni importanti o risultati già raggiunti.
  4. L'*usabilità* è la capacità del programma di consentire una interazione semplice, naturale ed efficace con l'utente finale. Per garantire questa capacità è necessario porre particolare attenzione alla definizione e alla realizzazione di una interfaccia amichevole, cioè di un insieme di metodi che facilitino l'interazione da parte dell'utente finale, e che si adattino alle sue caratteristiche.

- Le più importanti *qualità interne* sono: la riusabilità, la modularità, l'estendibilità, la portabilità e la compatibilità, la leggibilità e la capacità di verifica, la completezza ed efficacia della documentazione.

1. La *riusabilità* è la capacità del programma di essere riutilizzato, in tutto o in parte, per applicazioni diverse rispetto a quella per la quale è stato prodotto. Questa qualità è rivolta ad economizzare il processo di sviluppo dei programmi, sulla base del fatto che è spesso possibile individuare affinità concettuali tra diverse applicazioni. Una buona riusabilità dei programmi consente di beneficiare degli sforzi già compiuti per svilupparli, e di concentrarsi ogni volta sugli aspetti più innovativi di una applicazione. Ciò è tanto più importante se si pensa al costo di sviluppo e di mantenimento del software, che tende a diventare l'elemento predominante nell'ambito dei costi delle realizzazioni di sistemi informatici.
2. La *modularità* è la qualità relativa al grado di organizzazione interna del programma, e quindi al grado di strutturazione delle singole parti e della caratterizzazione del modo in cui esse cooperano per l'obiettivo generale. È evidente che la modularità riveste una importanza fondamentale quando il programma è di dimensioni ragguardevoli, ma è altrettanto chiaro che, indipendentemente da fattori quantitativi dell'applicazione, essa è comunque un elemento cruciale sia in fase di sviluppo del programma che in fase di valutazione del prodotto finale.
3. L'*estendibilità* è la capacità del programma di adattarsi facilmente a modifiche nei requisiti. Questa qualità è tanto più importante quanto più il programma è di dimensioni ragguardevoli, e quanto più l'ambiente applicativo in cui il programma opera è dinamico. Si noti che criteri di sviluppo modulari giocano un ruolo fondamentale per assicurare una buona estendibilità del programma.
4. La *portabilità* e la *compatibilità* sono relative alla facilità di trasferire il software prodotto in ambiti diversi rispetto a quelli previsti nella applicazione. Mentre la portabilità si riferisce in particolare alla possibilità di utilizzare il programma in ambienti hardware e software diversi da quelli per i quali è stato sviluppato, la compatibilità riguarda la capacità del programma di interagire in modo efficace con altri prodotti software. Come la riusabilità, anche queste qualità sono volte ad economizzare gli sforzi di sviluppo del software a fronte di eventuali cambiamenti dell'ambiente di elaborazione in cui esso opera.
5. La *leggibilità* e la *capacità di verifica* sono relative alla capacità del programma di esplicitare le scelte fatte in fase di progetto sia riguardo alla rappresentazione degli oggetti su cui esso opera, sia riguardo alle funzioni che questi oggetti manipolano. Si noti che la leggibilità è una qualità del programma scritto nel linguaggio di programmazione scelto, e non riguarda eventuali altri documenti a latere, che pure possono essere utilizzati come supporto per la documentazione e la spiegazione del programma.
6. La *completezza ed efficacia della documentazione* è la qualità relativa ai documenti di sviluppo e progetto (commenti nel programma, documentazione di progetto, documenti a latere del programma ecc.), che si accompagnano al programma codificato, e riguarda in particolare la loro capacità di esplicitare in modo completo e non ambiguo le scelte fatte durante il progetto, ed il ruolo svolto dalle varie componenti nell'ambito del programma complessivo.

1.4 Nota bibliografica

Per molti degli argomenti trattati abbiamo preso come riferimento [22] e [24], che consigliamo anche per ulteriori approfondimenti. Per quanto riguarda la qualità del software, rimandiamo a [43].

1.5 Esercizi

Esercizio 1.1 Considerare i seguenti contesti applicativi e ragionare sulla necessità di far ricorso per essi ad applicazioni sequenziali, concorrenti o dipendenti dal tempo.

1. sistema di controllo di una centrale nucleare;
2. sistema di prenotazione dei voli di un aeroporto;
3. risolutore di sistemi di equazioni;
4. sistema di interrogazione di una banca dati;
5. sistema operativo di un elaboratore elettronico.

Esercizio 1.2 Considerare le seguenti proprietà di un impianto Hi-Fi e stabilire quali di esse sono esterne e quali sono interne.

1. il tempo dedicato al collaudo;
2. la fedeltà sonora;
3. la probabilità di malfunzionamenti nel primo anno di utilizzazione;
4. la dimensione del trasformatore per l'alimentazione;
5. l'ergonomia dei comandi;
6. la linearità della risposta in frequenza.

Esercizio 1.3 Assimilando le qualità di un programma alle proprietà delle automobili, il fatto che i motori di un certo modello possono essere montati su diversi modelli di automobile a quali qualità fa riferimento?

Esercizio 1.4 Le qualità dei programmi non sono necessariamente indipendenti tra loro. Ad esempio, all'aumentare della modularità aumenta in genere anche la leggibilità. Considerare due qualità alla volta e valutare il loro rapporto reciproco.

Capitolo 2

La fase di progettazione

La fase di progettazione è la più critica di tutto il ciclo di sviluppo di un programma, e lo scopo principale di questo testo è l'approfondimento delle problematiche relative a questa fase. Questo capitolo illustra le nozioni fondamentali relative alla fase di progettazione, che si possono così riassumere:

- i *princìpi* di progettazione, che sono le linee guida generali che si devono seguire per produrre software di qualità;
- le *tecniche* di progettazione, che sono i metodi mediante i quali si produce il programma in modo coerente con i princìpi;
- gli *strumenti* per la progettazione, intesi come mezzi espressivi (ad esempio i linguaggi di programmazione) e ausili di varia natura utilizzati nella progettazione.

2.1 Princìpi di progettazione

I princìpi di progettazione sono le linee guida generali che dovrebbero costituire la base di ogni scelta e strategia di progetto. In questo paragrafo presentiamo le considerazioni sui princìpi limitando la discussione a tre di essi, che reputiamo i più significativi e importanti, e dai quali tutti gli altri possono essere in qualche modo derivati. Essi sono:

- distinzione tra il livello della *concettualizzazione* e il livello della *realizzazione*,
- uso dell'*astrazione*,
- *decomposizione* del problema e della soluzione in parti indipendenti.

2.1.1 Distinzione tra concettualizzazione e realizzazione

Il primo principio di progettazione di cui ci occupiamo suggerisce di considerare la fase di progettazione come composta di due sottofasi distinte:

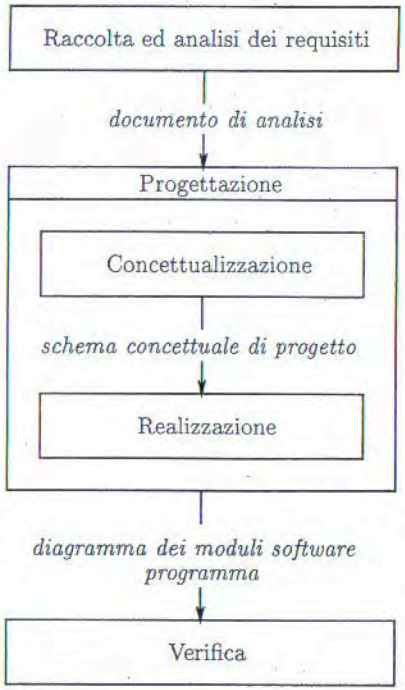

Figura 2.1 La struttura della fase di progettazione

- la concettualizzazione della soluzione, il cui prodotto è una descrizione astratta della soluzione del problema elaborativo, chiamata *schema concettuale di progetto*,

- la realizzazione in termini della tecnologia informatica, il cui prodotto è il programma redatto in un particolare linguaggio di programmazione.

La figura 2.1 mostra sinteticamente la struttura della fase di progettazione che risulta da questa suddivisione.

A parte la distinzione in due vere e proprie sottofasi, il principio di progettazione suggerisce di procedere durante tutto il progetto muovendosi su due piani distinti: il piano della concettualizzazione, in cui ci si concentra su *cosa* deve fare il programma, senza ancora affrontare il problema relativo a come il programma svolge le sue funzioni, ed il piano della realizzazione, in cui ci si concentra su *come* il programma realizza quanto deciso. Nel seguito di questo paragrafo discuteremo separatamente la concettualizzazione e la realizzazione.

Concettualizzazione

Appartengono alla fase di concettualizzazione tutte quelle attività relative al concepimento della soluzione informatica del problema, dalla specifica degli oggetti da manipolare, fino alla ideazione delle scelte algoritmiche su cui si baserà poi il programma finale.

Come si vede dalla figura 2.1, l'input alla fase di concettualizzazione è l'input all'intera fase di progettazione, e cioè il documento di analisi, prodotto nella fase di raccolta e analisi dei requisiti. L'output della concettualizzazione è lo schema concettuale di progetto, ovvero la *specifica* di cosa deve fare il programma, ancora espressa in modo indipendente dal linguaggio di programmazione. A livello generale, lo schema concettuale di progetto è costituito da:

- l'esplicitazione dei dati di interesse per il progetto dell'applicazione, e la specifica delle loro proprietà;

- l'esplicitazione delle operazioni di interesse per il progetto dell'applicazione, e la specifica dei corrispondenti dati di input e di output.

Il modo in cui lo schema concettuale viene prodotto, formalizzato e specificato dipende dalle tecniche e dalle metodologie usate. Illustreremo nei capitoli 9 e 10 un metodo generale per il disegno dello schema concettuale di progetto, basato sulla tecnica della modularizzazione.

È utile a questo punto chiarire maggiormente il rapporto tra la fase di raccolta e analisi dei requisiti e la fase di progettazione. Molte metodologie di analisi dei requisiti prevedono infatti di produrre qualcosa di simile a quello che abbiamo chiamato schema concettuale, ed è quindi importante sottolineare le differenze che, nella nostra accezione, hanno il documento di analisi e lo schema concettuale di progetto. Sebbene il documento di analisi contenga spesso, in qualche forma, la specifica dei dati e delle operazioni di interesse all'applicazione, si possono individuare le seguenti differenze con gli aspetti rappresentati nello schema concettuale:

1. Gli aspetti presenti nel documento di analisi sono spesso più ampi e generali rispetto a quanto rappresentato nello schema concettuale di progetto. Infatti, mentre nella fase di raccolta e analisi dei requisiti c'è l'esigenza di modellare in qualche forma l'organizzazione complessiva in cui si situa il sistema software, nella fase di progettazione ci si concentra sulle caratteristiche del sistema informatico da realizzare, ovvero su quella parte del sistema complessivo che verrà realizzata nel programma.

2. L'enfasi nel documento di analisi riguarda le proprietà generali dei dati, delle operazioni, dei flussi di dati e dei flussi di controllo, e non il dettaglio delle classi di dati e delle classi di operazioni da realizzare.

3. Nello schema concettuale, le categorie di rappresentazione sono orientate alla realizzazione informatica, ed hanno quindi caratteristiche che consentono la specifica dettagliata di cosa dovranno fare i programmi per realizzare correttamente gli elementi individuati.

La fase di concettualizzazione richiede cultura, disciplina, e, spesso, notevole creatività. Un prerequisito necessario per una buona riuscita di questa attività è ovviamente una comprensione completa e approfondita del problema in esame. Tuttavia, ciò può non essere sufficiente per dominare la vastità delle scelte da effettuare e la

complessità delle attività da condurre. Proprio per questo è praticamente impossibile fornire un elenco preciso di metodi e passi metodologici che consentano il successo di questa fase. Rimane infatti il problema che l'ideazione della soluzione è un processo in cui la creatività e l'intuizione possono giocare un ruolo fondamentale, se non predominante. Ciò nonostante, è possibile sia ragionare sulle attività mentali tipiche di questa fase, sia descrivere ed illustrare gli strumenti concettuali che possono essere di aiuto al progettista.

La fase di concettualizzazione si caratterizza in primo luogo come quella in cui viene effettuata la scelta della rappresentazione della realtà in gioco nel problema da risolvere. Una soluzione informatica di un problema può infatti considerarsi come la combinazione di una rappresentazione degli oggetti della realtà, e di una serie di manipolazioni simboliche che corrispondono al raggiungimento della soluzione. L'attività di rappresentazione coinvolge quindi due aspetti distinti e complementari: le informazioni e le operazioni che manipolano le informazioni. In entrambi questi aspetti giocano un ruolo basilare gli altri due princìpi di cui parleremo più avanti: l'astrazione e la decomposizione.

Consideriamo, ad esempio, un banale problema di ricerca dei numeri primi all'interno di una collezione di valori interi positivi. L'astrazione è quello strumento concettuale che ci consente di ignorare, ad un primo livello di dettaglio, l'organizzazione dettagliata dei dati in esame, di considerare in prima istanza la collezione degli elementi come un insieme senza struttura, e di concepire la soluzione come un'analisi degli elementi dell'insieme allo scopo di individuare tra essi quelli cercati, raccogliendoli in una seconda collezione. Già a questo livello di dettaglio, possiamo dire di avere ideato gli aspetti fondamentali della soluzione, ossia:

- la rappresentazione dei valori su cui il programma opererà mediante l'oggetto matematico "insieme",
- la realizzazione della soluzione algoritmica mediante l'operazione di analisi degli elementi dell'insieme,
- la rappresentazione del risultato, ancora mediante un insieme.

Si noti che, nelle considerazioni precedenti, abbiamo implicitamente assunto di avere a disposizione sia le operazioni fondamentali sugli insiemi (ad esempio l'analisi dei suoi elementi), sia un'operazione che, dato un valore intero, verifichi se esso sia primo. Abbiamo, cioè compiuto una strutturazione del nostro processo di concettualizzazione, decomponendo il problema in sottoproblemi, e astraendo dalle caratteristiche peculiari di questi sottoproblemi. È evidente che, al momento di dettagliare maggiormente la soluzione, ci occuperemo dei singoli sottoproblemi, ma lo faremo con la convinzione che, una volta trovata la soluzione ad essi, sappiamo come combinare queste soluzioni in modo corretto per ottenere il risultato globale.

Nell'esempio precedente sono presenti, sia pure in forma molto semplificata, gli altri princìpi fondamentali che riteniamo più rilevanti nella fase di concettualizzazione e che discutiamo nei prossimi due paragrafi: l'astrazione e la decomposizione. Ciò mette in evidenza che i princìpi di progettazione non sono tra loro indipendenti.

Realizzazione

Come già detto, intendiamo per realizzazione di un programma quel processo che, partendo dallo schema concettuale di progetto, permette di ottenere uno specifico programma che realizza gli elementi di tale schema. Ricordiamo che la fase successiva al progetto è la verifica, che ha lo scopo di stabilire, tra le altre cose, se il programma sia corretto.

La fase di realizzazione viene spesso vista come scomposta in due attività:

1. scelta dell'architettura del programma in termini di un insieme di parti indipendenti, dette *moduli software*;
2. realizzazione dei singoli moduli, e generazione del programma complessivo.

La prima attività riguarda la necessità di scomporre il sistema complessivo in un insieme di moduli, stabilire le relazioni tra essi, e individuare con precisione le funzionalità che tali moduli devono assicurare. La scelta su quali e quanti moduli realizzare viene fatta sulla base di un'analisi di come aggregare gli elementi dello schema concettuale in moduli software.

La seconda attività è invece una attività prettamente di programmazione, in cui i vari moduli vengono realizzati nel linguaggio di programmazione scelto, vengono provati singolarmente, e vengono poi integrati in un unico programma. Una buona realizzazione di un programma, dopo una buona concettualizzazione, è quindi legata al corretto uso del linguaggio di programmazione. Per ottenere questo risultato è necessario avere una buona conoscenza del linguaggio di programmazione, cioè interpretare ogni sua forma espressiva esattamente allo stesso modo della macchina, e far uso delle strutture del linguaggio per esprimere efficacemente quanto si è ottenuto con la concettualizzazione. L'attività di realizzazione sarà poi facilitata dalla capacità del linguaggio di programmazione di ridurre il lavoro intellettuale richiesto per realizzare il programma a partire dal risultato di una concettualizzazione. Chiameremo questa quantità di lavoro intellettuale *distanza cognitiva*. Per ridurre al massimo la distanza cognitiva, il linguaggio di programmazione deve permettere di realizzare programmi che abbiano una struttura simile alla struttura della concettualizzazione. L'attività di realizzazione si compie allora producendo, in corrispondenza di ogni elemento strutturale della concettualizzazione, una parte di un programma. Ogni parte del programma sarà così, in ogni momento, in grado di evocare l'elemento strutturale della concettualizzazione da cui è derivato. Questo aspetto rappresenta uno degli obiettivi seguiti nella definizione dei linguaggi di programmazione orientati agli oggetti, che analizzeremo in diversi capitoli di questo testo.

2.1.2 Astrazione

L'astrazione è un procedimento mentale che consente da una parte di evidenziare le caratteristiche pregnanti di una situazione o di un problema, e dall'altra di offuscare o addirittura ignorare gli aspetti che si ritengono secondari rispetto ad un determinato obiettivo. L'astrazione è un concetto molto generale, che comunemente viene utilizzato in tutte le attività mentali dell'uomo, e che si caratterizza in modo preciso quando si specializza ad attività informatiche.

Nell'ambito del progetto dei programmi, l'astrazione consente di giungere ad una soluzione in modo razionale e controllato. I processi di astrazione che in genere si effettuano nel progetto di programmi si possono così riassumere:

- Astrarre per *concettualizzare*: la differenza tra concettualizzazione e realizzazione si può spiegare con l'esigenza di procedere nel progetto dei programmi astraendo, ad un primo livello di dettaglio, dalle scelte che riguardano la programmazione in un particolare linguaggio. Di questo aspetto abbiamo già discusso nel paragrafo precedente.

- Astrarre per *decomporre*: questo meccanismo di astrazione è collegato ad uno dei fondamentali princìpi di progettazione, che discuteremo nel prossimo paragrafo.

- Astrarre per *rappresentare*: in questo caso, si utilizza l'astrazione per cogliere gli aspetti essenziali di una situazione del mondo reale, allo scopo di definire opportune strutture matematiche che colgano tali aspetti e siano al tempo stesso manipolabili da processi algoritmici. Ad esempio, se in un programma si devono gestire informazioni su una rete autostradale, sarà utile in fase di progetto fare riferimento ad una rappresentazione di tale rete attraverso una struttura di tipo grafo.

- Astrarre per *generalizzare*: in questo caso, si utilizza l'astrazione per produrre software che, partendo da una situazione particolare, sia generalizzabile e quindi riutilizzabile in altri contesti. Ad esempio, se dobbiamo gestire in un programma una matrice con 10 righe e 10 colonne, sarà utile scrivere il programma in modo indipendente dalle dimensioni della matrice, allo scopo di rendere più generale possibile la soluzione del problema in esame.

- Astrarre *sui dati*: in questo caso si utilizza il concetto di tipo astratto di dato per far riferimento a strutture matematiche costituite da valori ed operazioni su tali valori, senza considerare il modo in cui queste strutture sono organizzate e realizzate mediante costrutti di un linguaggio di programmazione. Nella fase di concettualizzazione, è molto importante saper utilizzare efficacemente l'astrazione sui dati, proprio perché lo scopo di questa fase è l'ideazione di uno schema di soluzione, e non la precisa formulazione della sua realizzazione informatica. È quindi cruciale saper ignorare gli aspetti che non giocano un ruolo fondamentale a questo livello, ed è ovvio che la precisa organizzazione dei dati è un aspetto che può essere trattato ed approfondito nelle fasi successive di progetto. D'altro canto, è altrettanto ovvio che una qualunque ideazione della soluzione non può spesso prescindere da una scelta di massima della rappresentazione degli oggetti che si devono manipolare. È quindi necessario che in fase di concettualizzazione si sappia selezionare la struttura astratta più idonea rispetto alle peculiarità del problema in esame. In questo può essere di grande aiuto una conoscenza approfondita delle strutture di dati, quali matrici, insiemi, alberi, grafi ecc., che più comunemente vengono utilizzate nelle soluzioni di problemi informatici e delle loro proprietà. È infatti la loro rappresentazione e la loro realizzazione che può essere ignorata in questa fase, ma non la loro utilizzazione. Ad esempio, se il problema si riferisce ad una situazione in cui le informazioni sono naturalmente strutturate in modo gerarchico, è ovvio che la concettualizzazione della soluzione potrà basarsi sulla scelta di organizzare i dati secondo un albero. Al contrario, se le relazioni tra gli oggetti in gioco non hanno una natura gerarchica, si potrà fare riferimento al semplice concetto di relazione. Se poi è naturale pensare direttamente in termini di una struttura di dati, sarà altrettanto naturale basarsi sulla scelta di considerare la relazione come un grafo, e ragionare direttamente su questa struttura.

  L'astrazione sui dati è un aspetto che sta ricevendo una attenzione particolare in questi anni, specialmente perché è quello più rilevante quando nella progettazione del software si privilegia l'orientazione ai dati (o agli oggetti). Proprio per l'importanza che l'astrazione sui dati riveste, noi approfondiremo le relative problematiche nella parte IV, in particolare per fornire maggiori elementi sul concetto stesso di tipo astratto di dati e sulla sua utilizzazione nelle fasi di concettualizzazione e di realizzazione.

- Astrarre *sulle funzioni*: in questo caso si sfrutta la possibilità di concentrare l'attenzione su *cosa* fa una particolare operazione, astraendo rispetto a *come* essa realizzi il suo compito. Durante la concettualizzazione, operiamo quindi una astrazione funzionale ogni volta che pensiamo ad una parte dell'algoritmo in modo astratto, specificandone il compito essenziale e le assunzioni sulla base delle quali questo può essere svolto, senza tenere conto delle modalità con cui il compito stesso può essere svolto e delle complicazioni che esso può presentare.

  L'astrazione funzionale ha un corrispettivo evidente in molte attività che conduciamo nella vita di tutti i giorni. Per fare un esempio banale, ogni volta che pianifichiamo gli impegni della nostra giornata (andrò alla posta a pagare la bolletta del gas, andrò a lezione all'università, pranzerò con Giulia, e tornerò a casa), noi astraiamo dalle modalità con cui svolgeremo le varie azioni, vedendole come funzioni ancora da dettagliare, e concentrandoci solo sulle loro interazioni reciproche.

2.1.3 *Decomposizione*

La decomposizione è un concetto molto semplice e intuitivo. Esso corrisponde all'attività mentale che viene svolta quando strutturiamo un problema o un sistema in parti, e stabiliamo precise relazioni tra esse. Non è facile fornire una serie di regole che stabiliscano precisamente il modo migliore di effettuare una decomposizione. In questo aspetto entrano in gioco la capacità creativa e la razionalità del progettista. È comunque possibile illustrare dei semplici criteri che possono essere utili specialmente al momento di valutare criticamente una scelta di decomposizione effettuata. La decomposizione di un problema dovrebbe godere di queste proprietà:

- Il livello di dettaglio dei sottoproblemi individuati deve corrispondere ad una precisa scelta progettuale e concettuale, e non venire come conseguenza di un'attività puramente intuitiva. È ad esempio importante valutare se la decomposizione che vogliamo effettuare mantenga o meno un omogeneo livello di astrazione e di complessità dei vari sottoproblemi, oppure se deliberatamente vogliamo scomporre il problema in modo che uno o più sottoproblemi mantenga un livello di complessità maggiore. In altre parole, è importante che il progettista abbia il completo controllo sulla comprensione dei sottoproblemi individuati, allo scopo di evitare che la complessità del problema rimanga nascosta in un solo sottoproblema, e che si ritrovi distribuita tra i sottoproblemi in modo prestabilito.

- Ogni sottoproblema deve effettivamente essere caratterizzabile in modo significativo ed essere risolvibile in modo indipendente; è evidente che se così non fosse, non avremmo ottenuto alcun risultato dalla decomposizione, e avremmo anzi complicato il problema e, corrispondentemente, la soluzione.

- Le soluzioni dei vari sottoproblemi devono essere combinabili in modo concettualmente semplice per formare la soluzione del problema originario. Questo criterio assicura che la decomposizione non abbia semplificato in modo sterile il problema originario, ma abbia effettivamente introdotto un elemento di strutturazione della soluzione con il quale poter controllare la complessità del problema e poter giungere più efficacemente alla soluzione.

Si noti che il processo di decomposizione può continuare in modo che i sottoproblemi individuati siano anch'essi ricorsivamente decomposti, secondo una *metodologia top-down*. D'altra parte, il processo di decomposizione può basarsi sull'idea di riutilizzare idee e scelte già adottate, magari in altri ambiti e applicazioni, e in questo caso si procede per decomposizione secondo una *filosofia bottom-up*. Ad esempio, dovendo affrontare il problema della ricerca dei numeri primi tra i dati memorizzati in una struttura gerarchica, si potrà cogliere l'analogia con il problema illustrato nel paragrafo 2.1, e si tenderà, analogamente a quanto fatto per quel caso, a decomporre il problema in modo che la verifica se un numero è primo sia un sottoproblema indipendente. Rispetto alla soluzione già trovata per il problema precedente, si cambierebbe così solo il metodo di analisi degli elementi, in particolare per adattarsi alla diversa organizzazione dei dati.

Si noti anche che pur avendo noi distinto astrazione e decomposizione, i due princìpi non sono tra loro indipendenti. Ad esempio, ogni volta che operiamo mediante l'astrazione sui dati stiamo anche implicitamente operando una decomposizione, in particolare enucleando le operazioni sui tipi di dato individuati, che potranno essere trattate indipendentemente dagli altri aspetti della soluzione. D'altro canto, l'astrazione gioca un ruolo fondamentale nella decomposizione, perché è astraendo dalle caratteristiche peculiari dei sottoproblemi, e concentrandosi sulla loro mutua interazione che operiamo una suddivisione del problema che goda delle proprietà viste prima.

La decomposizione è un concetto molto generale e può essere utilizzato per descrivere molte tecniche di progettazione dei programmi. Per mostrare un esempio, illustriamo come l'induzione possa essere considerata una manifestazione di questo principio. La decomposizione per *induzione* viene applicata ogniqualvolta si scompone un problema P in più parti in modo tale che una o più parti presentino gli stessi elementi concettuali di P. Si noti che parliamo di decomposizione per induzione quando ci vogliamo riferire ancora alla fase di concettualizzazione, in cui l'oggetto di interesse è l'algoritmo risolutivo di un problema, mentre l'analogo concetto in fase di realizzazione è la *ricorsione*, intesa come una particolare tecnica di programmazione che consente di tradurre in un programma un algoritmo concepito mediante la decomposizione per induzione.

La decomposizione per induzione si basa sul criterio di individuare diverse caratteristiche che le istanze di P possono evidenziare e, sulla base di queste caratteristiche, di procedere alla suddivisione in corrispondenti sottoproblemi in modo tale che:

1. esista almeno un sottoproblema B (base) che può essere risolto in modo diretto, secondo una soluzione concettualmente diversa rispetto alla soluzione pensata per P;

2. esista almeno un sottoproblema (induttivo) P' che ha esattamente le stesse proprietà rilevanti di P, anche se ha una struttura che, secondo un prefissato criterio di ordinamento, risulti minore di quella che caratterizza P, e che quindi può essere risolto mediante la stessa soluzione pensata per P. Il fatto che il sottoproblema P' evidenzi le stesse proprietà di P assicura che esso può essere risolto esattamente con lo stesso metodo scelto per risolvere P, mentre il fatto che P' evidenzi una struttura in qualche modo di dimensione minore di quella di P assicura che, eseguendo il procedimento su una istanza reale del problema P, la decomposizione giunga al caso in cui l'istanza considerata si riveli un caso particolare del sottoproblema B, e quindi il processo risolutivo possa avere termine.

2.2 Tecniche di progettazione

In questo paragrafo illustriamo brevemente una classificazione delle principali tecniche di progettazione, cercando di evidenziare quelle che stanno assumendo maggiore importanza alla luce delle recenti evoluzioni dei metodi di progettazione e programmazione.

- *Tecniche di specifica.* Abbiamo più volte detto che il prodotto della fase di concettualizzazione è il cosiddetto schema concettuale di progetto. Lo schema concettuale non è espresso in termini del linguaggio di programmazione, poiché descrive la soluzione ad un livello ancora indipendente dagli aspetti legati alla realizzazione. Al contrario, per descrivere lo schema concettuale si fa ricorso a metodi e formalismi ad hoc. Le tecniche di specifica sono appunto quelle tecniche che consentono di esprimere gli elementi essenziali di uno schema concettuale. Analizzeremo nella parte IV una di queste tecniche, che prevede di utilizzare un formalismo grafico per descrivere gli aspetti fondamentali dello schema concettuale, ed un linguaggio basato sulla logica e sull'algebra per descrivere gli aspetti concettuali dei tipi astratti di dato.

  I princìpi di progettazione si ritrovano nelle tecniche di specifica sotto diversi aspetti. Molte tecniche di specifica seguono esplicitamente il principio della decomposizione, fornendo strumenti per suddividere la specifica in parti e per stabilire opportune relazioni tra esse. Riguardo all'astrazione, l'idea stessa della specifica è una manifestazione dell'astrazione, perché induce una differenziazione evidente tra il cosa il programma deve fare rispetto al come lo fa.

- *Tecniche di programmazione.* Le tecniche di programmazione riguardano i metodi per la strutturazione e la stesura dei programmi a partire dallo schema concettuale. Negli anni '70 sono state studiate diverse tecniche di programmazione, tra le quali le tecniche di programmazione strutturata e di programmazione ricorsiva. In questo testo, assumiamo che il lettore abbia familiarità con queste tecniche. Più recentemente sono state introdotte altre tecniche di programmazione, come la programmazione logica e la programmazione orientata agli oggetti, legate a nuovi linguaggi di programmazione che si sono affermati in diversi contesti. Torneremo su queste tecniche nel prossimo paragrafo, quando illustreremo una classificazione dei linguaggi di programmazione. Sulla programmazione orientata agli oggetti svolgeremo poi diversi approfondimenti nelle parti del libro dedicate alla modularizzazione e alle tecniche algoritmiche.

  Molte tecniche di programmazione sono concrete manifestazioni di due dei princìpi di progettazione che abbiamo discusso. Ad esempio, la programmazione ricorsiva è la controparte, a livello realizzativo, della decomposizione per induzione. Un altro esempio è relativo alla programmazione orientata agli oggetti, che è nata con l'obiettivo di dotare i linguaggi di programmazione di espliciti meccanismi per realizzare l'astrazione sui dati.

- *Modularizzazione.* La tecnica della modularizzazione consente di razionalizzare lo sviluppo del software, costruendo programmi costituiti da parti indipendenti ed interagenti. Il termine *modulo* significa "parte" di programma, e viene poi istanziato in modo diverso in fase di concettualizzazione e in fase di realizzazione. A livello generale, un modulo è caratterizzato da una struttura interna, è definito con un determinato scopo, offre all'esterno un certo insieme prefissato di servizi utilizzabili da altri moduli, ed ha precise relazioni con altri moduli. L'uso dei

moduli permette di procedere nello sviluppo del programma decidendo prima l'architettura globale, e concentrandosi poi su ogni singolo modulo, sulle sue funzioni e sulle sue caratteristiche. Ciò favorisce uno sviluppo razionale anche se il programma è di grandi dimensioni.

La modularizzazione sta assumendo sempre più importanza nella progettazione dei programmi, specialmente in relazione alla sua rilevanza nella realizzazione di programmi riusabili. Per questa ragione, dedicheremo la parte IV del testo all'approfondimento di questa tecnica. Nella esposizione delle tematiche relative alla modularizzazione avremo modo di sottolineare quanto essa sia intrinsecamente coerente con i princìpi di progettazione: ad esempio, la modularizzazione è una concreta manifestazione del princìpio della decomposizione.

- *Tecniche di progettazione di algoritmi e strutture di dati.* Uno degli aspetti più importanti della progettazione dei programmi riguarda la definizione degli algoritmi e delle strutture di dati più appropriate per un determinato problema. Specialmente per le applicazioni più sofisticate, la progettazione dell'algoritmo risolutivo di un particolare problema determina diversi aspetti relativi alla qualità del prodotto finale, e richiede molta abilità ed esperienza. Un aspetto importante di questa esperienza riguarda da una parte la sensibilità a formalizzare un problema algoritmico, e dall'altra la conoscenza delle tecniche algoritmiche, che consentono di definire soluzioni efficaci ed efficienti per molte classi di problemi. Nella parte V entreremo nei dettagli delle tecniche algoritmiche. Nello studio delle tecniche algoritmiche avremo modo di apprezzare quanto stretto sia il loro rapporto con i princìpi di progettazione. Analizzeremo, ad esempio, la tecnica "divide et impera", che si basa sul princìpio della decomposizione, e approfondiremo gli aspetti relativi alla suddivisione tra concettualizzazione e realizzazione nella progettazione di algoritmi.

  Delle basi della progettazione di strutture di dati parleremo nella parte IV. Una trattazione approfondita di strutture di dati sofisticate esula dagli scopi di questo testo.

- *Tecniche di progettazione per specifiche classi di applicazioni.* Come abbiamo già detto, questo libro affronta le tematiche di base della progettazione, e si rivolge quindi, nel momento della esemplificazione, ad applicazioni tradizionali. Dobbiamo però notare che esistono diverse tecniche di progettazione specificamente definite per applicazioni particolari, quali quelle concorrenti, quelle dipendenti dal tempo, o quelle orientate alla gestione dei dati. Queste tecniche esulano dagli scopi di questo libro.

2.3 Strumenti di progettazione

Esistono diverse tipologie di strumenti utilizzabili nella progettazione del software. A livello generale, possiamo distinguere tra:

- strumenti di specifica per la concettualizzazione,
- strumenti di analisi per la concettualizzazione,
- strumenti per la programmazione.

Gli strumenti di specifica per la concettualizzazione sono quei formalismi (testuali, grafici) mediante i quali si descrive lo schema concettuale di progetto, e si specificano le caratteristiche di tutti gli elementi in esso rappresentati. Nei capitoli 9, 10 e 11 illustreremo diversi strumenti di questo tipo.

Gli strumenti di analisi per la concettualizzazione aiutano il progettista nell'analisi e nella verifica del documento di analisi e dello schema concettuale di progetto. Strumenti di questo tipo sono spesso dei sistemi software che consentono di eseguire controlli, automatici o semiautomatici, sullo schema concettuale di progetto, al fine di scoprire possibili errori, ridondanze, e incompletezze. Si usa spesso il nome di CASE (Computer Aided Software Engineering) per riferirsi a questo tipo di strumenti.

Gli strumenti per la programmazione sono quelli che supportano le fasi di scrittura del programma, compilazione e test. Lo strumento di primaria importanza in questa categoria è ovviamente il linguaggio di programmazione. Altri strumenti che tipicamente fanno parte di questa categoria sono:

- *Text editor*, che assistono il programmatore nella scrittura e nell'aggiornamento del programma. Particolare rilevanza hanno i cosiddetti *editor guidati dalla sintassi*, che sono in grado di specializzare l'attività di scrittura rispetto a uno specifico linguaggio di programmazione, segnalando o prevenendo errori sintattici, e proponendo opportune correzioni.

- *Compilatori*, che traducono un programma da linguaggio ad alto livello a linguaggio macchina.

- *Linker* (o *collegatori*), che consentono di creare un unico programma eseguibile a partire da più moduli compilati e da componenti di librerie.

- *Interpreti*, che eseguono le istruzioni di un programma senza esplicitamente passare per una fase di traduzione e di generazione del programma eseguibile.

- *Debugger simbolici*, che consentono al progettista di controllare l'esecuzione di un programma, eseguendo le istruzioni un passo alla volta, e fornendo la possibilità di esaminare e modificare lo stato del programma ad ogni passo.

Non è scopo di questo testo approfondire le problematiche relative agli strumenti. Poiché lo strumento essenziale della fase di progettazione è il linguaggio di programmazione, su questo strumento concentreremo la nostra attenzione nella parte restante di questo capitolo.

Molte delle considerazioni metodologiche che vengono svolte in questo libro sono largamente indipendenti dal tipo di linguaggio di programmazione impiegato nella realizzazione di un programma. Quando, però, si entrerà maggiormente in dettaglio sugli aspetti tecnologici dello sviluppo dei programmi, dovremo necessariamente svolgere considerazioni più concrete, riferendoci ad un particolare linguaggio, il C++. Allo scopo di connotare in modo preciso le caratteristiche del C++ in relazione agli altri linguaggi, vogliamo dapprima delineare una classificazione dei linguaggi attualmente disponibili, e in seguito svolgere alcune considerazioni generali e preliminari sul paradigma della programmazione orientata agli oggetti, che permea diversi aspetti importanti del linguaggio C++.

2.3.1 Una classificazione dei linguaggi di programmazione

I linguaggi di programmazione attuali vengono schematicamente classificati in:

- *linguaggi imperativi*, caratterizzati dalla filosofia che un programma è la specifica di un insieme di istruzioni che corrispondono a precisi comandi impartiti ad una macchina, che, una volta attivata, li esegue pedissequamente allo scopo di ottenere la soluzione ad un dato problema;

- *linguaggi di programmazione funzionale*, caratterizzati dall'assunto che un programma è la specifica di una funzione, che, sulla base di un insieme di dati di ingresso, calcola il risultato secondo una legge specificabile in modo matematico;

- *linguaggi di programmazione logica*, caratterizzati dalla filosofia che un programma è la specifica di una relazione che sussiste tra un insieme di dati, e tale specifica è costruita mediante un sistema formale basato sulla logica matematica;

- *linguaggi di programmazione orientata agli oggetti*, caratterizzati dalla filosofia che un programma è la specifica di un insieme di oggetti che rappresentano gli elementi della situazione in gioco in un certo problema, e ciascun oggetto è specificato in termini di una struttura e di un insieme di operazioni tramite le quali si ottiene il comportamento voluto per risolvere il problema.

È opportuno sottolineare che la suddetta classificazione non corrisponde in modo netto e preciso né ad una tassonomia fondata su basi teoriche, né alle caratteristiche effettive dei linguaggi esistenti ed utilizzati. In uno stesso linguaggio possono infatti convivere caratteristiche che in genere si ascrivono a diverse classi di linguaggi. Così, ad esempio, esistono proposte di linguaggi funzionali che hanno strutture di rappresentazione orientate all'uso di oggetti, oppure linguaggi imperativi che offrono meccanismi propri della programmazione funzionale, o ancora, linguaggi imperativi con costrutti linguistici che consentono di definire classi e loro proprietà, e di strutturare il programma secondo una filosofia orientata agli oggetti. Ciò nonostante, la classificazione introdotta è utile perché fornisce un semplice mezzo per individuare le caratteristiche più pregnanti dei vari linguaggi.

Come già detto, in questo libro faremo riferimento in particolare al linguaggio C++, che appartiene alla classe dei linguaggi imperativi con caratteristiche di orientazione agli oggetti. La scelta di questo linguaggio deriva da diverse considerazioni. La prima riguarda il fatto che i linguaggi per la programmazione imperativa allo stato attuale sono i linguaggi di programmazione più utilizzati nelle applicazioni industriali. La seconda considerazione è che il C++ ha diverse caratteristiche interessanti per produrre programmi con orientazione agli oggetti, e allo stesso tempo supporta in qualche misura anche lo stile di programmazione funzionale. Infatti, in C++ si può in linea di principio produrre un programma che di fatto è costituito da espressioni che invocano funzioni e che vengono valutate in fase di esecuzione. La terza considerazione riguarda la previsione di una diffusione sempre crescente del C++ negli ambienti di produzione del software.

2.3.2 *Introduzione alla programmazione orientata agli oggetti*

Il termine "orientato agli oggetti" (*object-oriented*) ha recentemente avuto una grandissima diffusione in molti campi dell'informatica. Questo termine viene infatti usato in diversi contesti (ad esempio, basi di dati, ed interfacce utente) per descrivere software di varia natura costruito secondo la filosofia descritta nel precedente paragrafo

È impossibile dare una definizione univoca e precisa di cosa si intenda per orientazione agli oggetti. Informalmente possiamo dire che un programma orientato agli

oggetti è costituito da un insieme di oggetti che interagiscono tra loro, ogni oggetto essendo composto da uno stato interno costituito dai dati associati all'oggetto, ed un insieme di operazioni che modificano tale stato. Il punto centrale della programmazione è quindi l'oggetto, in opposizione alla programmazione tradizionale in cui l'attenzione maggiore è rivolta alle procedure (e funzioni) che manipolano i dati. Più precisamente, si parla di progettazione e programmazione con:

- *orientazione alle funzioni* quando il concetto primario nell'attività di progettazione o programmazione corrisponde ad un'operazione necessaria al processo risolutivo del problema; gli oggetti manipolati dall'operazione sono considerati, in questo caso, parti componenti del concetto;

- *orientazione agli oggetti* quando il concetto primario nell'attività di progettazione o programmazione corrisponde ad una astrazione sugli oggetti della realtà in gioco; questa astrazione è ottenuta classificando gli oggetti stessi sulla base delle operazioni che li manipolano; in questo secondo caso, allora, le operazioni che si applicano agli oggetti di una classe sono parti componenti dell'astrazione.

Per meglio apprezzare le caratteristiche della programmazione agli oggetti, è opportuno descrivere sinteticamente alcuni concetti essenziali che connotano tale filosofia. Nel seguito di questo paragrafo descriveremo sinteticamente tali concetti, che possiamo riassumere in:

- oggetti, classi ed incapsulamento,

- *information hiding* (dall'inglese, nascondere l'informazione),

- ereditarietà.

Ritorneremo su questi concetti quando parleremo delle tecniche di progettazione, ed in particolare di modularizzazione. Prima di iniziare la descrizione, vogliamo però sottolineare che quando si parla di *linguaggi* orientati agli oggetti (ad esempio, EIFFEL, SMALLTALK e C++), ci si riferisce al fatto che tali linguaggi forniscono dei costrutti linguistici per supportare la programmazione orientata agli oggetti. Rimane però valida l'osservazione che molte delle caratteristiche di un programma dipendono dal metodo con cui è stato progettato, piuttosto che dal linguaggio in cui viene espresso: è ad esempio possibile scrivere programmi con orientazione agli oggetti in linguaggi quali il PASCAL (che non supporta specifici costrutti per gli oggetti) e scrivere programmi assolutamente non orientati agli oggetti in C++.

Oggetti, classi ed incapsulamento

Il principio fondamentale dell'orientazione agli oggetti è l'*incapsulamento*. Col termine incapsulare intendiamo l'assemblamento in un'unica entità (l'oggetto) di dati e operazioni. In che forma i dati e le operazioni vengono incapsulati negli oggetti dipende dallo specifico linguaggio di programmazione utilizzato.

In C++ l'incapsulamento viene supportato tramite il costrutto di *classe*. Una classe è un tipo di dato, simile ad un tipo record, i cui valori sono oggetti descritti mediante un insieme di campi, dove ciascun campo rappresenta

- o una proprietà associate all'oggetto, specificata mediante il valore di un dato,

- oppure una funzione applicabile sull'oggetto.