Capitolo 8
Design e programmazione object-oriented

Si è parlato di design object-oriented. Senza la pretesa di trattare il tema in modo approfondito, vediamo ora alcuni concetti della programmazione a oggetti che vanno considerati e che una buona conoscenza dei pattern permette di sfruttare al meglio.

È difficile trovare una definizione standard sui requisiti minimi affinché un linguaggio possa essere considerato object-oriented. Lo dimostra il fatto che alcuni autori considerano come linguaggi che permettono la programmazione a oggetti anche il primo Ada e Modula-2.

Ci interessano qui unicamente le caratteristiche minime che ci permetteranno di capire e discutere le strutture dei design pattern.

Information hiding e data abstraction

Sono i primi concetti che hanno portato alla programmazione a oggetti. Con *information hiding* si intende il nascondere i dettagli dell'implementazione di un programma o di una parte di esso. L'utilizzo di queste componenti avviene attraverso operazioni definite in un'interfaccia. Il concetto di *data abstraction* ("astrazione sui dati") si basa sul primo e significa che la semantica

di un tipo di dati viene definita unicamente attraverso le funzioni che hanno accesso ad essi. La rappresentazione del tipo rimane perciò nascosta.

Ereditarietà

Si tratta della possibilità di specificare gerarchie di classi, in cui le sottoclassi sono specializzazioni delle classi che si trovano ai livelli superiori (superclassi). Le classi ereditano variabili e metodi. Possono inoltre estendere e modificare il comportamento acquisito dalla superclasse.

Composizione

Diversamente dal meccanismo di ereditarietà, qui si definisce una nuova classe come la somma (composizione) di altre classi. Si tratta della relazione *has-a*, contrapposta alla relazione *is-a* dell'ereditarietà.

Polimorfismo

Il polimorfismo, visto da un punto di vista pratico, è quel meccanismo che permette di riferirsi a un oggetto attraverso qualsiasi variabile che abbia la stessa *interfaccia*. Questo ha conseguenze anche nel meccanismo di passaggio di parametri. Nel caso object-oriented, se una funzione definisce di accettare come parametro un oggetto di una certa classe, significa che a quella funzione si potranno passare anche istanze di sue sottoclassi

Binding dinamico

Viene anche chiamato *late binding*. Significa che l'associazione tra il nome di un metodo e il suo indirizzo in memoria viene effettuata durante l'esecuzione del programma, non durante la compilazione o il link/load.

Information hiding e data abstraction

Sono tra i primi concetti che hanno portato alla programmazione OO. Information hiding e data abstraction si possono realizzare in modo elegante già con linguaggi di programmazione come Modula-2 e Ada.

Si tratta di poter utilizzare una certa struttura di dati senza dover conoscere i dettagli della sua implementazione. L'utilizzo avviene attraverso la definizione di funzioni di interfaccia (funzioni dichiarate nel *definition module* per Modula-2, oppure funzioni dichiarate *public* in C++ e Java) che permettono l'accesso ai dati gestiti nella struttura stessa.

I dati rimangono completamente nascosti in Modula-2, che per questo distingue due file: uno per la definizione delle funzioni di interfaccia e uno per la definizione della struttura di dati e l'implementazione delle funzioni. In C++ la struttura di dati è visibile nella classe, ma non accessibile direttamente, perché non dichiarata *public*. I dati possono essere letti solo attraverso appositi metodi pubblici, la cui implementazione può comunque trovarsi separata dalla definizione della classe.

In Java, invece, non c'è nessuna separazione fisica all'interno di una classe. Tutti i dati e i metodi sono definiti nello stesso file. Il concetto di *hiding* è perciò puramente legato ai modificatori utilizzati per i metodi: tutto ciò che è public può essere considerato interfaccia, il resto consiste invece in metodi utilizzati per scopi interni alla classe (private), interni al package o a eventuali sottoclassi (protected). La separazione fisica tra implementazione e interfaccia in Java si realizza con l'uso dell'elemento interface.

Ereditarietà

Un concetto importante quando si lavora con le classi è l'ereditarietà. Questa permette di specificare nuove classi partendo da altre già definite, ereditando metodi e variabili. L'ereditarietà ha permesso di rendere reali nella programmazione i concetti di estendibilità e riutilizzo.

Con linguaggi non OO, il riutilizzo del software viene risolto con le librerie di funzioni. In realtà è però più frequente il caso in cui solo una parte di funzioni può essere utilizzata così come era stata prevista dal programmatore della libreria, mentre altre dovrebbero essere modificate o completamente riscritte.

Con l'ereditarietà, estendere codice diventa più semplice perché non è necessario copiare vecchie righe di programma: basta ereditare e specificare le nuove caratteristiche desiderate. Ogni classe eredita automaticamente le proprietà e le funzionalità delle classi che le stanno sopra nella gerarchia (superclassi).

Supponendo che esista una classe Bicicletta e che si desiderasse creare una nuova classe BiciDaCorsa con caratteristiche e funzionalità aggiuntive (come la variabile peso e il relativo metodo getPeso()), ecco in figura 8.1 come si presenterebbe la nuova classe:

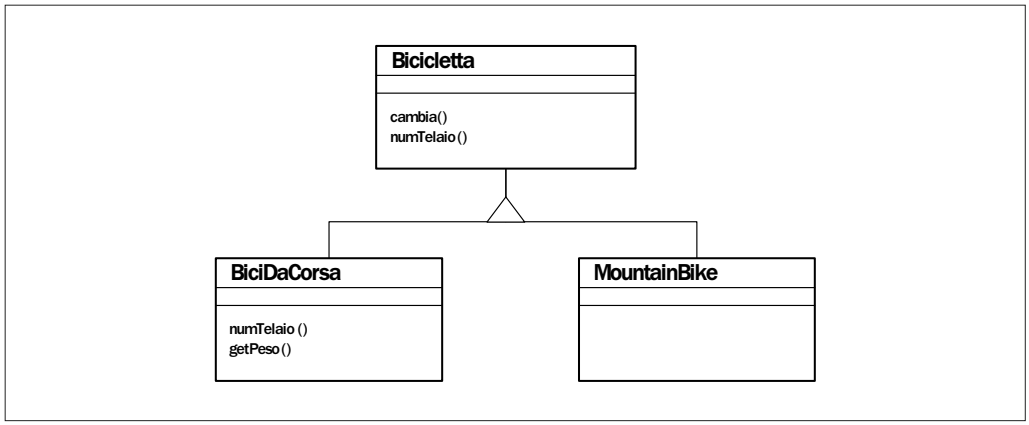

Figura 8.1 – Schema di ereditarietà.

```
public class BiciDaCorsa extends Bicicletta
{
  float peso;

  public BiciDaCorsa(int marce, String numeroTelaio, float p){
    super(marce, numeroTelaio);
    peso = p;
  }

  public float getPeso(){
    return peso;
  }
}
```

Trovare la giusta gerarchia di classi diventa fondamentale per l'eliminazione di ridondanze. Quando si definiscono nuove classi è importante trovare quali sono gli eventuali elementi in comune per racchiuderli in una o più superclassi.

Però attenzione: il cambiamento di una superclasse modifica automaticamente anche il comportamento delle sue sottoclassi. Vedremo che tra classe padre e classe figlio si crea una dipendenza a volte indesiderata (alto valore di accoppiamento).

Ogni classe è in grado di utilizzare le variabili ad essa associate e le variabili delle sue superclassi (alcuni linguaggi, come Java, permettono a una classe di nascondere anche alle sottoclassi, se lo desidera, le sue variabili, dichiarandole `private`).

Una cosa simile si ha con i metodi. Qui però è possibile "ridefinire" (*overriding*) il metodo di una superclasse, ridefinendolo nella sottoclasse.

C'è comunque sempre la possibilità che il metodo di una sottoclasse chiami, al suo interno, il suo metodo corrispondente nella superclasse (quello con il suo stesso nome e gli stessi parametri, cioè il metodo che ha ridefinito). Questo si verifica perché può capitare che il nuovo metodo della sottoclasse debba semplicemente estendere il funzionamento del metodo già esistente.

Ereditarietà e leggibilità

Capita che le ragioni per scegliere di creare una relazione di ereditarietà tra due classi siano più tecniche che coerenti con il problema che si sta tentando di modellare.

L'utilizzo dell'ereditarietà a scopo "tecnico" serve senza dubbio a condividere comportamento e a minimizzare codice e variabili, evitando inutili e pericolose ridondanze. D'altra parte, proprio perché non usato necessariamente per modellare la logica del problema, rende il codice meno leggibile. Soprattutto in una prima fase di modellizzazione, non è sempre consigliabile organizzare le classi gerarchicamente solo perché condividono comportamento.

Un esempio tipico, anche se un po' riduttivo, di gerarchia che distoglie dal dominio del problema è quella tra rettangolo e quadrato. Il quadrato è un caso particolare di rettangolo, ci si aspetterebbe quindi che sia l'eventuale classe `Quadrato` a dover ereditare da `Rettangolo` (in questo modo il

design sarebbe chiaro). La logica dell'ereditarietà come mezzo di condivisione del comportamento inviterebbe però a fare esattamente il contrario. Per gestire un quadrato ho unicamente bisogno dell'informazione riguardante un lato. Per gestire un rettangolo ho invece bisogno di due lati, base e altezza. La tentazione potrebbe essere quella di ereditare un lato dal quadrato e poi aggiungere il secondo lato nella sottoclasse Rettangolo.

Motivi per l'ereditarietà

Ci sono vari motivi per utilizzare l'ereditarietà. Uno di questi è la specializzazione della classe di base, come nell'esempio di Bicicletta appena mostrato. Qui una classe concreta estende un'altra classe concreta. Chiamiamo classe concreta una classe dalla quale possono essere creati oggetti, in contrapposizione con classe astratta.

Ma ci sono vari altri motivi per ricorrere all'ereditarietà: implementazione (una classe concreta viene creata come estensione di una classe astratta), limitazione (la nuova classe ha un utilizzo dei metodi più limitato rispetto alla classe di base), combinazione (ereditarietà multipla). Il concetto di ereditarietà multipla è abbastanza critico, perché realizzato in modo diverso nei diversi linguaggi.

La filosofia usata in Java, il linguggio di riferimento per tutti gli esempi del libro, è quella secondo cui l'ereditarietà multipla complica il linguaggio e il suo utilizzo in modo superiore ai vantaggi che comporterebbe. Inoltre la possibilità di avere ereditarietà multipla invita a complicare il design, con rischi di anomalie, invece di obbligare alla semplicità.

La possibilità di gestire un oggetto concreto attraverso diversi punti di vista, dato dall'ereditarietà multipla, viene però assicurato in Java dal concetto di *interface*, una sorta di classe astratta con solo dichiarazioni di metodi. Una nuova classe Java può ereditare da un'unica classe (gerarchia di implementazione), ma può implementare più *interface* (gerarchia di *interface*). Importante: con l'elemento *interface* viene mantenuto il concetto di polimorfismo, che vedremo in seguito.

Composizione

Si tratta di un concetto ben presente da prima della programmazione a oggetti e della conseguente suddivisione del codice in classi.

La composizione è però di fondamentale importanza anche nella programmazione OO; ce ne renderemo conto ben presto, quando entreremo più nel dettaglio del discorso relativo ai pattern.

Quando si vogliono creare classi da altre classi, non sempre l'eredità (relazione *is-a*) è la soluzione migliore. Esiste un'altra forma di relazione tra classi: la composizione (relazione *has-a*), qui trattata in modo generico come associazione tra due classi.

Riprendiamo l'esempio precedente e pensiamo a una possibile implementazione della classe Bicicletta, supponendo che esistano elementi complessi (non primitivi) di tipo Telaio e Ruota:

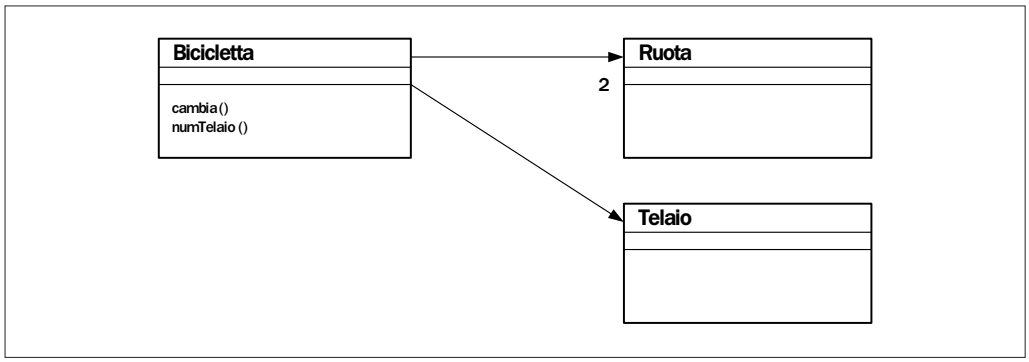

Figura 8.2 – Schema di composizione.

```
public class Bicicletta
{
  Telaio telaio;
  Ruota rAnteriore, rPosteriore;
  int marce;

  public Bicicletta(int m, String numeroTelaio){
    telaio = new TelaioInCarbonio(numeroTelaio);
    rAnteriore = new RuotaLenticolare();
    rPosteriore = new RuotaLenticolare();
    marce = m;
  }

  int getMarce(){
    return marce;
  }

  String numTelaio(){
    return telaio.getNumero();
  }
  ...
}
```

Una bicicletta, oltre alle informazioni sulle marce, definiti con tipi primitivi, ha anche un telaio, definito con la classe Telaio e due ruote, specificate attraverso due oggetti della classe corrispondente. Ogni oggetto della classe Bicicletta contiene quindi riferimenti a oggetti di altre classi.

Per questioni di semplicità consideriamo la composizione come l'unica relazione *has-a*, anche se si potrebbero distinguerne diversi tipi, a seconda della granularità d'intenti che permette il linguaggio (in Java esiste un unico modo, con riferimento, mentre ad esempio in C++ è possibile legare due oggetti sia attraverso una variabile di tipo puntatore all'elemento, che con una variabile dell'elemento stesso). Una distinzione può essere fatta anche tenendo conto della dipendenza a runtime tra gli oggetti: se elimino un oggetto Bicicletta cosa succede al suo aggregato oggetto Telaio? Dev'essere eliminato anche lui? Viene condiviso con altri oggetti Bicicletta? Si parla allora di associazioni, aggregazioni, *acquaintance* ("conoscenza") e così via, che qui trattiamo sotto lo stesso cappello: composizione.

Visibilità

In realtà questo tipo di relazione è abbastanza evidente: un oggetto è composto da una serie di altri oggetti. Vale però la pena di sottolinearlo perché offre alcuni interessanti vantaggi rispetto alla stessa relazione di eredità. Con la composizione vediamo così la seconda modalità che ci permette il riutilizzo del software.

L'ereditarietà permette di definire l'implementazione di una classe utilizzandone un'altra. Questo tipo di riutilizzo viene anche chiamato *white-box reuse*, perché la sottoclasse ha una certa visibilità di alcuni dettagli della classe da cui ha ereditato. Inoltre la classe che eredita non è protetta dai cambiamenti della classe genitore. Potenzialmente, ogni modifica alla classe padre comporta una revisione di tutte le sottoclassi.

La composizione permette di assemblare diversi tipi di funzionalità per crearne una nuova più complessa. Se gli oggetti utilizzati per la composizione utilizzano il concetto di information hiding, si ottiene un riutilizzo di codice che può essere definito *black-box reuse*, perché nessun dettaglio degli oggetti utilizzati è visibile.

Entrambi i metodi hanno vantaggi e svantaggi. Da un punto di vista del software design, quando entrambe le soluzioni possono essere adottate, sembra comunque preferibile la seconda, sia per la maggiore eleganza del *black-box reuse*, sia perché garantisce una maggiore flessibilità. La composizione può essere modificata a runtime, perché realizzata a livello di oggetto; la gerarchia no.

Polimorfismo e binding dinamico

I concetti visti in precedenza non avrebbero così grossa importanza se non venissero associati a quelli di polimorfismo e binding dinamico (anche chiamato *late binding*), quest'ultimo in contrapposizione al binding statico (anche chiamato *early binding*).

Polimorfismo

Esistono varie forme di polimorfismo. A noi interessa la definizione che ci permetterà di capire e sfruttare al meglio i design pattern.

Il polimorfismo è quel meccanismo che permette di riferirsi a un oggetto attraverso qualsiasi variabile che abbia la stessa *interfaccia*. Per variabile con la stessa interfaccia si intende, in questo caso specifico, una variabile definita dello stesso tipo o del tipo di una sua classe superiore.

```
Bicicletta bici = new BiciDaCorsa();
```

oppure il già visto

```
Telaio telaio = new TelaioInCarbonio();
```

Nell'esempio che segue, l'argomento bici, dichiarato di tipo Bicicletta, in realtà può essere un oggetto della classe Bicicletta, come definito, ma anche uno della classe BiciDaCorsa o uno di una ipotetica classe MountainBike, anch'essa specificata come sottoclasse di Bicicletta.

Ciò significa che il metodo controllaBicicletta() accetta come parametro qualsiasi elemento scelto all'interno dell'intera gerarchia Bicicletta.

```
void controllaBicicletta(Bicicletta bici){
  ...
}
```

Binding

Quando si ha a che fare con linguaggi di programmazione procedurali non a oggetti, non capita quasi mai di dover distinguere diversi *binding*, cioè i modi usati per associare le funzioni alle loro chiamate: tutte le chiamate dirette a funzione sono infatti già note al momento della compilazione, e sono poi in grado di eseguire il *link*. Si parla in questo caso di binding statico.

Per un linguaggio orientato a oggetti questa è una restrizione troppo grande, perché obbliga il programma a conoscere esattamente le funzioni che verranno chiamate e i loro rispettivi indirizzi prima che il programma stesso venga eseguito.

Completiamo l'esempio appena visto:

```
void controllaBicicletta(Bicicletta bici){
  System.out.println("No di telaio: " + bici.numTelaio());
}
```

Visto che l'argomento può contenere riferimenti a oggetti di diverso tipo, non sempre è possibile sapere prima del runtime, a quale metodo numTelaio() l'oggetto bici si riferisce.

Il messaggio inviato può corrispondere a metodi distinti: numTelaio() di Bicicletta, numTelaio() di BiciDaCorsa, e così via, quindi il compilatore non può sapere in anticipo quale variante verrà chiamata. Utilizzerà allora il binding dinamico, cioè l'indirizzo del metodo da chiamare verrà risolto durante l'esecuzione, a dipendenza del tipo di oggetto che invoca la chiamata (che manda il messaggio).

Puntatori a funzione

In realtà un funzionamento che ricorda il polimorfismo e che necessita di binding dinamico è già presente in linguaggi di programmazione procedurali (Modula-2, C), ma in casi particolari: quando si usano puntatori a funzione.

Durante la compilazione non è possibile prevedere quale funzione verrà chiamata attraverso il puntatore, perciò viene creata una tabella di indirizzi. L'indirizzo giusto verrà scelto solo in runtime. Il puntatore a funzione è quindi una tecnica con una potenzialità che va oltre ai classici esempi mostrati nei libri di programmazione procedurale. Non rappresenta solo la possibilità di passare una funzione come argomento di un'altra.

D'altro canto, passare una funzione come argomento in un linguaggio come Java è possibile solo associando la funzione a un oggetto, passando poi l'intero oggetto. Questa idea è alla base di un pattern molto usato, il pattern Command. Questo modello, proprio perché fa uso di un oggetto per racchiudere un metodo, dà anche la possibilità di mantenere uno stato (variabili di istanza).

È importante a questo punto notare che in Java i concetti di polimorfismo e binding dinamico, così come appena mostrati, sono possibili in gerarchie di classi, ma anche in gerarchie di interfacce. In altre parole, per sfruttare polimorfismo e binding dinamico non è indispensabile avere gerarchie di implementazione (classi): bastano gerarchie di design (*interface*), che oltretutto permettono l'ereditarietà multipla. Posso cioè riferirmi ad un oggetto attraverso una delle sue interfacce, esattamente come posso farlo attraverso una sua superclasse.

Verifica dei tipi

L'idea di binding dinamico ha anche altre sfaccettature. Una di queste è il controllo durante la compilazione. Pur parlando comunque di binding dinamico, si distingue tra controllo statico e controllo dinamico.

Il controllo statico è quello presente in Java. Serve ad evitare errori a runtime, perché assicura che un certo oggetto è in grado di rispondere a un messaggio.

In `controllaBicicletta()` l'argomento `bici` potrà chiamare `numTelaio()` perché questo metodo fa parte dell'interfaccia di `Bicicletta` (e quindi anche di `BiciDaCorsa` e di tutte le classi che si trovano nella gerarchia di `Bicicletta`). Il compilatore esegue questo controllo e non permette chiamate a metodi non definiti nell'interfaccia in questione, così come non permette che venga associato a una variabile di tipo `Bicicletta` un oggetto che non faccia parte della gerarchia.

Un'assegnazione di questo tipo sarebbe sbagliata:

```
Bicicletta bici = new TelaioCarbonio()   // errore in compilazione!
```

Con il controllo dinamico non ci sono invece controlli sulle interfacce e sui metodi che elementi con queste interfacce sono in grado di chiamare. La responsabilità della correttezza è a carico del programmatore. Eventuali errori si hanno solo a runtime. Questo metodo è quello utilizzato da linguaggi dinamici come CLOS e Smalltalk, che si prestano bene per la prototipazione rapida. Anche l'assegnazione di un oggetto a una variabile è libera: non c'è il

controllo dei tipi. A runtime interessa unicamente che l'oggetto in questione sappia rispondere ai messaggi che gli vengono inviati. Il tutto è molto più libero e i programmatori Smalltalk, con orgoglio, si riferiscono a questo come al *true dynamic binding*.

Tipi e classi

Parlando di classi, polimorfismo e binding dinamico, diventa utile la distinzione tra tipo e classe. Per non complicare le cose e perché il linguaggio usato negli esempi di pattern che vedremo in seguito è Java, utilizzeremo la distinzione applicata a questo linguaggio.

Una classe può essere considerata l'implementazione di un tipo. Definisce il modo in cui un oggetto viene implementato: lo stato interno e la codifica dei metodi.

Un tipo è una caratterizzazione astratta di proprietà di un insieme di oggetti. Definisce l'interfaccia (con `interface` o con `class`) di un oggetto e le richieste a cui questo oggetto è in grado di rispondere.

Un oggetto può avere più tipi. Inoltre oggetti di classi diverse possono appartenere allo stesso tipo.

Concetto di riutilizzo

Ci sono vari elementi che favoriscono il riutilizzo. A parte la standardizzazione e la fiducia (la mancanza di fiducia spinge all'implementazione di soluzioni proprie…) consideriamo tre elementi come fondamentali per il riutilizzo effettivo di software:

Adattabilità

Si tratta della possibilità di creare nuove soluzioni senza intervenire nel codice esistente. È anche lo scopo ultimo della programmazione a oggetti: fare in modo che una modifica si riduca a una semplice aggiunta. Visto che l'architettura iniziale non può essere in grado di prevedere tutte le possibili modifiche che interverranno nel ciclo di vita di un programma, i passaggi di refactoring servono anche a questo: trasformare un programma per fare in modo che la prossima modifica di funzionalità diventi un'aggiunta al programma stesso.

Localizzazione

Si intende centralizzare tutto ciò che dev'essere adattabile all'interno del programma. Significa che tutto il resto è sufficientemente generico da poter essere riutilizzato in altri ambiti. Un esempio tipico di elementi da centralizzare sono i metodi factory (dal pattern Factory Method, che vedremo più avanti). La localizzazione è anche determinante per una buona manutenzione.

Astrazione

Questo è senza dubbio l'aspetto più importante se guardiamo al riutilizzo dal punto di vista dei design pattern. Le soluzioni astratte sono più facilmente adattabili. I design pattern sono infatti soluzioni astratte, modelli, che vanno utilizzati e riutilizzati in ambiti completamente

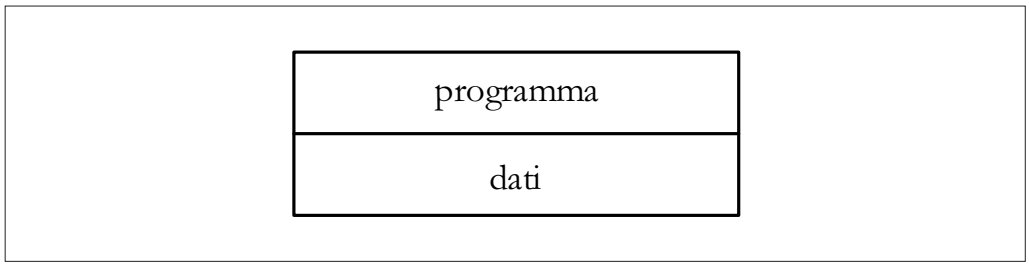

Figura 8.3 – Struttura iniziale.

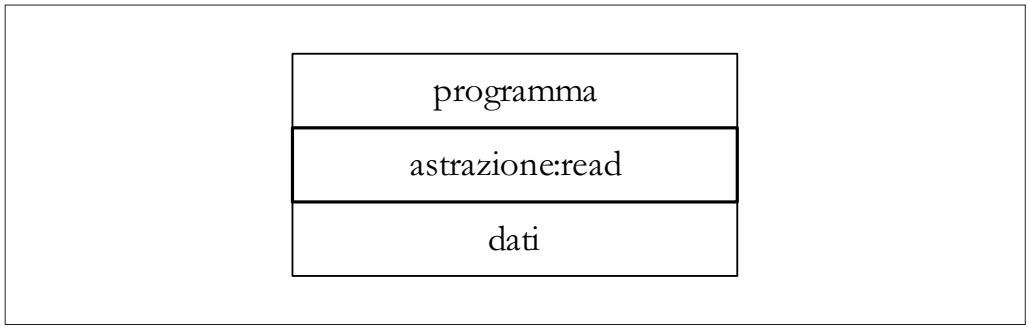

Figura 8.4 – Struttura con livello di astrazione sulla lettura dei dati.

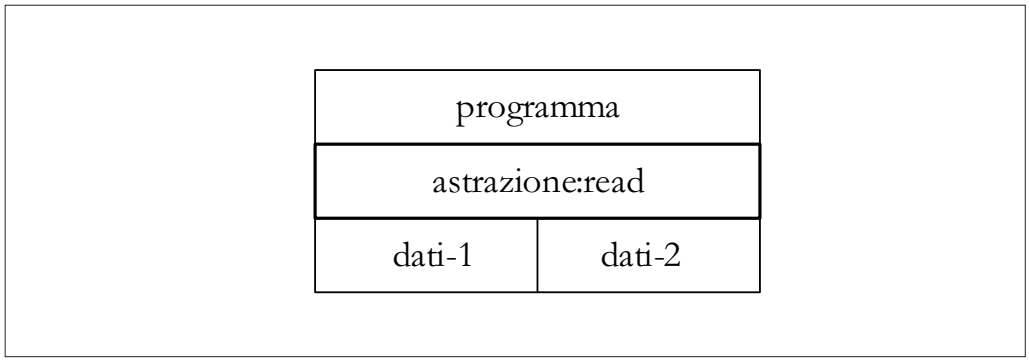

Figura 8.5 – Aggiunta di nuovi formati.

diversi. L'astrazione favorisce il riutilizzo perché è l'elemento alla base dell'estensibilità del software. Spesso un livello in più di astrazione è sufficiente per fare di una soluzione unica un modulo flessibile e pronto per nuove estensioni.

Consideriamo il seguente esempio: abbiamo un programma che legge dei dati da file system e questi dati si trovano in un determinato formato, vengono letti e interpretati dal programma principale che li usa all'interno della sua business logic. La struttura iniziale viene presentata nella figura 8.3.

Cosa succede se vogliamo modificare il formato dei dati nel file? Dove vanno effettuate le modifiche? È chiaro che la logica del programma dovrebbe rimanere indipendente dal formato del file. La soluzione per garantire indipendenza diventa allora quella di un livello di astrazione che crei un cuscinetto tra programma e dati e separi le due entità, come mostrato in figura 8.4.

In questo modo si raggiunge la genericità all'interno del programma, perché la modifica del formato dei dati, che a questo punto diventano interscambiabili, non causa nessun adattamento nel programma principale, cioè nella parte logica (figura 8.5).

In questo capitolo…

In questo capitolo abbiamo rivisto alcuni concetti basilari della programmazione a oggetti utili per la comprensione dei pattern. Vedremo ancora in seguito quanto sia importante la buona comprensione di uno di tali aspetti fondamentali, il polimorfismo, elemento che sta alla base di quasi tutti i modelli di design.