CSELT Torino, 01/10/98

NOTA INFORMATIVA

Design Patterns e Idiomi Java

Ennio Grasso
CSELT
email: ennio.grasso@cselt.it
http://andromeda.cselt.it/users/g/grasso/index.htm

Executive Summary

La fase di design di un sistema è quella fase cruciale che si interpone tra la fase di analisi e di implementazione. Durante il design vengono introdotti nuovi oggetti, la distribuzione di responsabilità, le relazioni e le regole di collaborazione tra gli oggetti di analisi possono essere modificate per rispettare nuovi vincoli talvolta contrastanti: incapsulamento, granularità, dipendenza, flessibilità, prestazioni, evoluzione, riuso, ecc. Ma una cosa sembra certa: un designer con esperienza riesce a fare un buon design perché applica l'esperienza per riusare soluzioni sperimentate con successo nel passato per risolvere problemi simili.

Per cercare la soluzione a un problema il cervello identifica delle strutture simili, dei **patterns** ricorrenti. Le pagine di questa nota sono una mia interpretazione dei patterns del famoso libro di Gamma [GOF1] in ambito Java, sottolineando gli idiomi particolari di questo linguaggio. Si può considerare un "Bignami" di [GOF1]. I requisiti per leggere questa nota sono una conoscenza dei principi dell'object-orientation in generale e di Java in particolare. Una po' di pratica di programmazione è di sicuro aiuto perché affrontare i patterns avendo già una qualche esperienza permette di riconoscerli più facilmente. Sapete infatti qual'è stata la mia reazione quando per la prima volta mi sono avvicinato ai concetti dei patterns? "Bhè cos'è sta roba? Queste cose le ho già viste diverse di volte, ma non ho mai dato tutta questa enfasi!". Ma il punto è proprio questo, i patterns non hanno la pretesa di inventare nulla. Sono solo una descrizione di pratiche di buon senso che normalmente si imparano dopo lunghi tirocini di esperienza. Un pattern non è mai inventato, ma solo riconosciuto, estrapolato e classificato. Il processo è induttivo: se mi accorgo di usare una certa soluzione per risolvere problemi simili (almeno un certo numero), questa soluzione può essere analizzata, estrapolata dal contesto per renderla astratta e più facilmente applicabile in altri domini, e infine catalogata, dando nome, problema, soluzione e conseguenze.

È un problema di design

La fase di design di un sistema è quella fase cruciale che si interpone tra la fase di analisi e di implementazione. Sebbene la fase di analisi possa essere affrontata con successo applicando principi metodologici, la fase di design tende a sfuggire ad ogni tentativo di inquadramento canonico. Durante il design vengono introdotti nuovi oggetti, la distribuzione di responsabilità, le relazioni e le regole di collaborazione tra gli oggetti di analisi possono essere modificate per rispettare nuovi vincoli talvolta contrastanti: incapsulamento, granularità, dipendenza, flessibilità, prestazioni, evoluzione, riuso, ecc. Definire un insieme di regole certe per il design è pressoché impossibile.

E se fare design è difficile, ancora di più è fare un “buon” design. Cosa si intende per “buon” design? Condizione necessaria è che risolva il problema, ma non basta. Dalla teoria della computabilità

sappiamo che esistono infiniti programmi che risolvono lo stesso problema: qual'è il migliore? Il giudizio è empirico e rientra in parte nel giudizio estetico degli esseri umani: così come diciamo "quella è una bella ragazza" o "questa è una bella casa", analogamente diciamo "questo è un programma fatto bene" perché risolve il problema in modo elegante, con meno linee di codice, con meno uso di risorse, perché si presta a essere facilmente modificato e riusato in altri contesti. Si potrebbe quasi dire che il design è una forma d'arte, che richiede doti difficili da catturare e razionalizzare. Ma una cosa sembra certa: **un designer con esperienza riesce a fare un buon design perché applica l'esperienza per riusare soluzioni sperimentate con successo nel passato per risolvere problemi simili**.

Un cervello a patterns

La deduzione logica e altri modelli formali sono modi possibili per risolvere problemi, ma applicabili in casi ideali dove si conoscono tutte le premesse. In molti casi reali per risolvere un problema si parte da dati incompleti, parziali e talvolta contraddittori dove un approccio ortodosso di un modello formale non funzionerebbe. In questi casi l'esperienza è l'arma più importante. Perché per risolvere un problema di analisi, o un esercizio di ricerca sugli alberi (e passare l'esame) occorre fare molti esercizi sull'argomento, che magari differiscono di poco? Se il cervello funzionasse in modo deduttivo, basterebbe dare le regole del dominio e applicarle per arrivare alla soluzione. In realtà è necessario fare esercizi per "allenare il cervello" a identificare soluzioni simili in modo che il giorno dell'esame l'esercizio da risolvere abbia "similitudini" con problemi già risolti e ci faciliti la soluzione.

Il cervello identifica in modo inconscio delle strutture simili, dei **patterns** ricorrenti. La parola "pattern" è difficile da tradurre in italiano perché significa modello, esempio, campione, ma ha anche il significato di avere una certa struttura interna. Christofer Alexander è l'inventore del concetto di pattern applicandolo nel campo dell'architettura. Secondo Alexander: "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to this problem in such a way the you can use this solution a million times over, without ever doing it the same way twice". Quindi **un pattern serve a risolvere un problema ricorrente documentando una soluzione provata con successo diverse volte**.

I patterns non hanno la pretesa di inventare nulla. Sono solo una descrizione di pratiche di buon senso che normalmente si imparano dopo lunghi tirocini di esperienza. Un pattern non è inventato, ma riconosciuto, estrapolato e classificato. Il processo è induttivo: se mi accorgo di usare una certa soluzione per risolvere problemi simili, questa soluzione può essere analizzata, estrapolata dal contesto per renderla astratta e più facilmente applicabile in altri domini, e infine catalogata, indicando nome, problema, soluzione e conseguenze. In questo processo di astrazione, la descrizione del pattern diventa sufficientemente generica perché rappresenti la soluzione a una famiglia di problemi. Un pattern funziona come uno "stampo" in cui le sue componenti sono istanziate nei vari casi concreti.

Design patterns

Il concetto di pattern corrisponde al modo con cui funziona il cervello e travalica ogni dominio particolare e quindi nel dominio dello sviluppo software ha senso parlare di **design patterns**. Tra i primi a applicare il concetto di pattern al design a oggetti individuando i patterns più importanti sono stati Gamma, Helm, Johnson, Vlissides (detti GOF = Gang Of Four) [GOF1] con il libro "Design Patterns, Elements of Reusable Object-Oriented Software", considerato la bibbia sui patterns. Il libro [GOF1] è soprattutto un catalogo di 23 patterns, ciascuno descritto sottolineando quattro elementi essenziali:

1. **Nome**. Sembra ovvio, ma dare nomi significativi ai patterns è molto importante. Un nome è un modo immediato e veloce per individuare un problema di design. Avere questi nomi nel proprio vocabolario permette di comunicare con altri designers passando complesse soluzioni di design con il semplice scambio di una parola. Dire "qui ho usato il *Singleton* perché dovevo essere sicuro che la risorsa venisse allocata una sola volta" comunica immediatamente la soluzione al problema e il problema stesso.
2. **Problema**. Per definizione un pattern serve per risolvere un problema.

3. **Soluzione**. Descrive la struttura del pattern: gli elementi partecipanti e le forme di collaborazione e interazione tra questi. La descrizione è sempre sufficientemente astratta da essere applicabile in diversi casi e situazioni reali.
4. **Conseguenze**. Ogni pattern, e quindi ogni modo per risolvere un problema, ha vantaggi e svantaggi. Proprio perché esistono infinite soluzioni a un problema, a seconda delle criticità certe volte un pattern è più adatto di un altro. Identificare esplicitamente le conseguenze di un pattern aiuta a determinare l'efficacia dello stesso nei vari casi.

Pur rimanendo sufficientemente generali e indipendenti dal linguaggio, i patterns [GOF1] vengono descritti con esempi C++ e Smalltalk. Le pagine di questa nota sono una mia interpretazione dei patterns [GOF1] in ambito Java, sottolineando gli idiomi particolari di questo linguaggio. Considerate questa nota un "Bignami" di [GOF1]. I requisiti per leggere questa nota sono una conoscenza dei principi dell'object-orientation in generale e di Java in particolare. Una po' di pratica di programmazione è di sicuro aiuto perché affrontare i patterns avendo già una qualche esperienza permette di riconoscerli più facilmente.

A cosa servono i patterns

1. Introducono un vocabolario comune. La conoscenza dei designers esperti non è organizzata secondo le regole sintattiche di un linguaggio di programmazione, ma in strutture concettuali più astratte. I patterns di design offrono un vocabolario comune per comunicare, documentare ed esplorare alternative di design. Permettono di descrivere un sistema in termini più astratti che non le semplici righe di codice del linguaggio di programmazione. I designers possono comunicare con frasi del tipo: "usiamo l'Observer in questo caso", o "fattorizziamo uno Strategy da queste classi". Avere un vocabolario comune evita di dover descrivere un'intera soluzione di design: basta nominare i patterns usati.
2. Permettono di capire più facilmente il funzionamento dei sistemi a oggetti. Molti sistemi a oggetti complessi usano design patterns. Conoscere i patterns aumenta la comprensione dei sistemi perché risultano più chiare le scelte di design.
3. Accelerano la fase di apprendimento. Diventare esperti designers richiede molto tempo. Il modo migliore è lavorare a lungo con sistemi a oggetti e osservare molti sistemi fatti da designers esperti. Avere conoscenza dei concetti dei patterns riduce il tempo necessario per diventare esperti.
4. Sono un complemento alle metodologie a aggetti. Le metodologie a oggetti cercano di risolvere la complessità del design standardizzando il modo con cui si affronta il problema. Ogni metodologia introduce una notazione (es. Object Modeling Technique) per modellare i vari aspetti del design e un insieme di regole che indicano come e quando usare ogni elemento della notazione. L'approccio ortodosso e standardizzato delle metodologie non è mai riuscito a catturare l'esperienza dei designers. I patterns sono un complemento, il tassello mancante alle metodologie a oggetti. In particolare, il passaggio dalla fase di analisi a quella di design è sempre la più critica. Molti oggetti di design non hanno una corrispettiva entità nel dominio di analisi e i patterns sono un mezzo essenziale per spiegare questi oggetti.
5. Permettono di anticipare i cambiamenti. Lo sviluppo del software passa in genere lungo tre fasi: *prototipazione*, *espansione* e *consolidamento*. Nella fase di prototipazione viene definita l'applicazione in modo che rispetti i requisiti iniziali. Quando l'applicazione viene messa in esercizio possono nascere nuovi requisiti che richiedono l'estensione delle funzionalità offerte: è la fase di espansione. Il problema è che spesso le funzionalità da introdurre richiedono grossi interventi sul software perché non flessibile per accettare i cambiamenti. Ecco che si entra nella fasi di consolidamento in cui il design viene rivisto per accomodare la flessibilità necessaria. Il consolidamento comporta separazione di classi in sotto componenti, muovere operazioni lungo la gerarchia di ereditarietà, razionalizzare le interfacce delle classi, ecc. Il processo di consolidamento è inevitabile e in genere costoso a meno che il design iniziale venga fatto in ottica evolutiva e i patterns aiutano in questo senso.

Premessa

Prima di tuffarci direttamente sui patterns, è opportuno riflettere su alcuni concetti generali dell'object-orientation per capire meglio il contesto del discorso che verrà fatto in seguito.

Librerie, Frameworks, Patterns, Idiomi

Vale la pena chiarire brevemente le differenze tra questi concetti.
Una **libreria** è un insieme di classi base che forniscono funzionalità generiche, es. liste, dizionari, ecc.
Un **framework** è un insieme di classi cooperanti che formano design concreto e riusabile per uno specifico dominio (es. interfacce grafiche). Il framework definisce l'architettura dell'applicazione e cattura le decisioni di design comuni nel dominio in cui si applica. A differenza di una libreria dove l'applicazione specifica il flusso principale del programma e richiama le classi di supporto, con un framework il flusso principale è già definito dal framework per cui è sufficiente specializzare il comportamento di alcune sotto componenti.
I **patterns** differiscono in almeno tre aspetti dai frameworks:

- sono concetti astratti mentre i framework sono reali. I frameworks in genere usano diversi patterns nella loro architettura.
- hanno una struttura molto più semplice e focalizzata a un particolare problema di design. I frameworks hanno una struttura complessa tipicamente composta da diversi patterns.
- sono generici e non legati a un particolare dominio. I frameworks sono sempre legati a un dominio particolare.

Sia patterns che **idiomi** descrivono soluzioni a problemi ricorrenti di design, ma mentre nei patterns soluzione e problema sono sufficientemente generici da essere indipendenti dal linguaggio di programmazione, gli idiomi fanno riferimento alle proprietà specifiche di un certo linguaggio e descrivono come implementare le soluzioni di design nel linguaggio.

Classi e interfacce

Nell'object-orientation c'è grande differenza tra classi e interfacce. Una classe definisce come un oggetto è implementato. Un'interfaccia definisce il tipo dell'oggetto, ossia cosa rappresenta. Un client che usa un oggetto è solo interessato alle sue interfacce e non a come sono implementate.
In Java, e in C++ con le classi astratte, un oggetto può avere diverse interfacce così come una stessa interfaccia può essere supportata da diverse classi, ossia avere diverse implementazioni.
Allo stesso modo è importante distinguere tra i concetti di ereditarietà e di sottotipo. In linguaggi come il C++, e alle volte anche in Java, l'ereditarietà e il sottotipo sono spesso confusi perché utilizzano lo stesso costrutto del linguaggio (**extends** in Java) anche se in realtà l'obiettivo è ben diverso. L'ereditarietà serve per riusare il codice e quindi agisce a livello di classe. Il sottotipo permette di sostituire una classe che supporta un certa interfaccia al posto di un'altra classe che supporta la stessa interfaccia o un suo super tipo.

Classi astratte e interfacce

Altra distinzione si ha tra interfacce e classi astratte. Chi proviene dal C++ e passa a Java spesso si domanda: "perché Java ha sia il concetto di interfaccia che di classe astratta, non ne basterebbe uno?". Classi astratte e interfacce hanno obiettivi diversi e sebbene in C++ non esista distinzione a livello di linguaggio, questa esiste sicuramente a livello di design. Tutti sanno che per avere il potere espressivo della Macchina di Touring basta le funzione zero, incremento e ricorsione. Ma tutti sanno che linguaggi più evoluti aiutano. Quindi che differenza c'è tra interfacce e classi astratte in Java?
Si **usa un'interfaccia** quando si deve definire il tipo di una classe identificando le operazioni offerte e astraendo da come sono implementate. Il polimorfismo viene sfruttato per dare comportamenti diversi alla stessa operazione.
Si **usa una classe astratta** quando in un design si vogliono fattorizzare alcuni comportamenti a livello di super classe, mentre altri comportamenti devono essere ridefiniti. Questa super non è comunque sufficiente a caratterizzare entità concrete per cui istanze di tale classe non devono poter essere create.

Ereditarietà e composizione
Le due tecniche più importanti per il riuso sono l'ereditarietà e la composizione. L'ereditarietà permette di riusare l'implementazione di una classe estendendo e modificando parte del suo comportamento. La composizione consiste nell'assemblare oggetti tra loro per ottenere funzionalità più complesse. Si può pensate alla relazione di ereditarietà come essere: "un gatto è un animale", mentre la composizione come avere: "un'auto ha un motore".
L'ereditarietà è più facile da usare e da tenere sotto controllo, ma è meno flessibile perché lega staticamente le classi tra loro. La composizione è più flessibile ma è difficile da governare e mantenere in modo pulito. Come linea generale bisognerebbe favorire comunque la composizione all'ereditarietà perché in generale la flessibilità ottenuta ripaga la complessità aggiuntiva. Questa però è solo un'euristica generale e non una regola fissa.

Polimorfismo implicito e parametrico
Se composizione e ereditarietà sono tecniche di riuso del codice, polimorfismo implicito e parametrico sono i due modelli per il riuso delle funzionalità.
Nei linguaggi a oggetti, la relazione di ereditarietà a livello di interfacce introduce una relazione di sottotipo tra le interfacce detta *polimorfismo implicito*. In alcuni linguaggi, come ML e C++, esiste un'altra forma di polimorfismo detta *polimorfismo parametrico* in cui è possibile definire esplicitamente delle *variabili di tipo*, ossia variabili che possono assumere valori nel dominio dei tipi. Ad esempio possiamo definire un algoritmo che data una lista di tipo *T*, dove *T* è una variabile di tipo e una funzione *Comparator: TxT→boolean*, esegue l'ordinamento della lista. L'algoritmo è generico perché può essere usato per liste di interi, stringhe, oggetti di tipo *User*, ecc. Il C++ è uno dei pochi linguaggi che supporta sia la programmazione generica con i *templates* che quella puramente a oggetti. Ma l'uso combinato di entrambi i modelli è spesso fonte di complessità e di difficile comprensione. Ciò che spesso accade è la tendenza a privilegiare un design a oggetti dove l'aspetto parametrico è mantenuto al minimo, oppure un design basato su templates e in questo caso è l'aspetto object-oriented a essere minimale. Java (come Smalltalk) non risente della mancanza di un supporto alla programmazione generica perché tutte le classi ereditano da *Object* per cui il solo polimorfismo implicito è in genere sufficiente a realizzare designs generici e riusabili.

Design Patterns e idiomi Java
Iniziamo dunque la rassegna dei patterns e idiomi Java. Il mio approccio è stato quello di prendere il catalogo [GOF1] e reinterpretarlo in ottica Java sottolineando di volta in volta gli idiomi particolari di questo linguaggio.

Patterns di creazione
Questi patterns aiutano a rendere un sistema indipendente da come gli oggetti sono creati, composti e rappresentati.

Singleton
È il pattern più semplice e serve quando si vuole che **esista una e una sola istanza di una certa classe**.

Il concetto chiave del Singleton è prevenire la possibilità di creare oggetti di una certa classe tramite il costruttore di new. L'idioma Java per implementare il pattern del Singleton prevede di dichiarare tutti i costruttori **privati** con almeno un costruttore esplicito altrimenti il compilatore genera un costruttore di default. Occorre inoltre avere:
- una variabile **privata statica** della classe che rappresenta l'unica istanza creata;
- un metodo **pubblico** `getInstace` che torna l'istanza. Questa potrebbe essere creata all'inizio o la prima volta che si richiama `getInstance`.

Siccome Java permette la clonazione degli oggetti, per completare l'idioma la classe dovrebbe essere dichiarata **final** con l'effetto di impedire la clonazione delle sue istanze. Se invece la classe

eredita da una gerarchia che implementa l'interfaccia *Clonable* occorre ridefinire il metodo `clone` e sollevare l'eccezione *CloneNotSupportedException*:

```
final class Singleton {

      private static Singleton instance = new Singleton();
      public static Singleton getInstance() {return instance;}
      private Singleton() {}

      public Object clone() throws CloneNotSupportedException {
            throw new CloneNotSupportedException();
      }
      public void method1() {...}
      public void method2() {...}
      ...
}
```

Factory Method

Serve quando si vuole **creare un oggetto di cui staticamente si conosce l'interfaccia o la super classe mentre la classe effettiva viene decisa a runtime**.

Il pattern del Factory Method risolve il problema sfruttando il polimorfismo del linguaggio.

Una tipica struttura a framework usa classi astratte e interfacce per definire e mantenere relazioni tra oggetti. Il framework è spesso anche responsabile per la creazione di questi oggetti. Ma un framework, in quanto tale, è soggetto a essere specializzato all'interno di applicazioni che forniscono sottoclassi di interfacce e classi astratte definite nel framework. Ma come fa il framework a istanziare oggetti di sottoclassi che non conosce? Il Factory Method risolve il problema. La struttura di questo pattern prevede:

- *AbstractObject* è tipicamente un'interfaccia ma può anche essere una classe;
- *ConcreteObject1,...,ConcreteObjectN* sono classi che implementano *AbstractObject*;
- *Creator* è una classe con un metodo **astratto** `createObject` che torna un oggetto di tipo *AbstractObject*;
- *Creator1,...,CreatorN* sono sottoclassi di *Creator*. Esiste una classe *CreatorN* per ogni classe *ConcreteObjectN* che implementa il metodo `createObject` istanziando proprio un oggetto *ConcreteObjectN.*

Se da un lato *Creator* e *AbstractObject* fanno parte del framework, *CreatorN* e *ConcreteObjectN* sono le rispettive specializzazioni per una certa applicazione. In questo modo il framework può creare oggetti di classe *ConcreteObjectN* senza previa conoscenza di questa classe. Il framework invoca `createObject` e grazie al polimorfismo l'effetto è quello di richiamare la versione specializzata del metodo nella classe *CreatorN*, la quale crea un'istanza di *ConcreteObjectN* e la restituisce con upcasting alla sua interfaccia (o super classe) *AbstractObject*. Questo è un aspetto importante perché il framework non conosce la classe *ConcreteObjectN*, ma ancora grazie al polimorfismo ogni invocazione sull'oggetto *AbstractObject* in realtà richiama l'implementazione definita da *ConcreteObjectN.*

```
// the framework
interface AbstractObject {
      public void aMethod();
}

abstract class Creator {
      abstract AbstractObject createObject();

      doSomething() {
            AbstractObject o = createObject();
```

```
            o.aMethod();
      }
}

// the application
class ConcreteObject1 implements AbstractObject {
      public void aMethod () {...}
}

class Creator1 extends Creator {
      AbstractObject createObject() {
           return (AbstractObject)new ConcreteObject1();
}
.....
// somewhere in the code
Creator c = new Creator1();
```

Il pattern del Factory Method ha lo svantaggio che per ogni classe *ConcreteObjectN* occorre definire una sottoclasse *CreatorN* corrispondente.

Prototype

serve negli stessi casi del **Factory Method**, ossia quando si vuole **creare un oggetto di cui staticamente si conosce l'interfaccia o la super classe mentre la classe effettiva viene decisa a runtime**.

Il pattern del Prototype risolve il problema sfruttando la meta informazione del linguaggio.

Lo svantaggio del Factory Method è che occorre definire una classe *CreatorN* che implementa il metodo `createObejct` per ogni nuova classe *ConcreteObjectN*. Se il numero di classi *ConcreteObject1,..., ConcreteObejctN* è molto grande, o se le classi sono aggiunte a runtime tramite *dynamic-class-loading*, allora il Factory Method non è più adatto. In Java è possibile sfruttare Java Reflection per avere accesso alle informazioni di meta livello che riguardano la classe di un oggetto e decidere la sua istanziazione. I partecipanti al pattern del Prototype sono:

- *AbstractObject* è tipicamente un'interfaccia ma può essere una classe qualunque;
- *ConcreteObject1,...,ConcreteObjectN* sono classi che implementano *AbstractObject*.
- *Creator* è una classe con un metodo `createObject` che torna un oggetto di tipo *AbstractObject*;

La differenza ora è che non occorre una classe *CreatorN* per ogni classe *ConcreteObjectN*. Il metodo `createObejct` di *Creator* sfrutta la meta informazione per istanziare l'oggetto. L'idioma del Prototype in Java prevede di passare come parametro il meta oggetto che rappresenta la classe che si vuole istanziare.

```
class Creator {
      AbstractObject createObject(Class c) {
            return (AbstractObject)c.newInstance();
      }
}
```

Nell'esempio sopra è stata usata la versione semplice per l'istanziazione di oggetti che abbiano un costruttore di default senza parametri. Se invece si vuole richiamare un costruttore con parametri basta ottenere un oggetto *Constructor* dall'oggetto *Class*.

```
class Creator {
      AbstractObject createObject(Class c) {
            Class[] args = {...} // type of the arguments
            Constructor ctor = c.getConstructor(args);
```

```
            return (AbstractObject)ctor.newInstance();
      }
}
```

Una variante dell'idioma consiste nel generalizzare ulteriormente il metodo `createObject` passando non il meta oggetto che rappresenta la classe bensì il nome della classe. In questo modo la classe può essere caricata dinamicamente rendendo ancora più flessibile il programma perché il nome della classe, in formato stringa, può essere ricavato nei modi più disparati (passato come parametro, comunicato da remoto, ecc.)

```
class Creator {
      AbstractObject createObject(String className) {
            Class c = Class.forName(className);
            return (AbstractObject)c.newInstance();
      }
}
```

Una variante ulteriore fa ricorso alla clonazione degli oggetti invece della meta informazione. In [GOF1], infatti, il pattern del Prototype viene descritto in questa variante perché in tal modo è applicabile in linguaggi come il C++ privi di meta livello. In questa variante il *Creator* offre un metodo aggiuntivo per registrare dinamicamente dei prototipi (da qui il nome del pattern), ossia dei rappresentanti degli oggetti da creare. I prototipi sono inseriti in un dizionario e riferiti tramite una chiave, ad esempio un nome. Il metodo `createObject` riceve come parametro la chiave dell'oggetto che si vuole istanziare, recupera il prototipo dal dizionario e richiama il suo metodo `clone`.

```
class Creator {
      private Hashtable protos = new Hashtable();
      void registerPrototype(String key, Object proto) {
            protos.put(name, proto);
      }
      AbstractObject createObject(String key) {
            proto = (AbstractObject)protos.get(key);
            if (proto == null) return;
            return (AbstractObject)proto.clone();
      }
}
```

Abstract Factory

Definisce **un'interfaccia per creare istanze di classi dipendenti tra loro senza conoscenza della loro classe effettiva**.

I patterns del **Factory Method** e del **Prototype** permettono di creare un oggetto di cui staticamente si conosce l'interfaccia o la super classe mentre la classe effettiva viene decisa a runtime. Il pattern dell'Abstract Factory risolve lo stesso problema ma in questo caso esiste una dimensione aggiuntiva. Invece di una sola interfaccia o super classe *AbstractObject* ci sono diverse classi che in qualche modo hanno una dipendenza reciproca. Capita spesso di definire strutture di classi dipendenti tra loro, ad esempio classi che rappresentano prodotti di una stessa famiglia. In questi casi è opportuno raccogliere i metodi per la creazione di istanze di queste classi in un'unica interfaccia. I partecipanti al pattern sono:

- *AbstractObject1,..., AbstractObjectN* sono tipicamente interfacce ma possono anche essere classi e rappresentano la famiglia di classi;
- *ConcreteObject1,..., ConcreteObjectN* sono classi che implementano rispettivamente *AbstractObject1,...,AbstractObjectN*;
- *AbstractFactory* è l'interfaccia che raccoglie i metodi di creazione della famiglia di oggetti;
- *ConcreteFactory* implementa *AbstractFactory* specificando il comportamento dei metodi di creazione.

```
interface AbstractFactory {
      public AbstractObject1 createObject1();
      ...
      public AbstractObjectN createObjectN();
}

class ConcreteFactory implements AbstractFactory {
      public AbstractObject1 createObject1() {
            // use the Factory Method or the Prototype
            return (AbstractObject1)new ConcreteObject1();
      }
      ...
}
```

Notate che il pattern dell'Abstract Factory sia in realtà una generalizzazione dei patterns Factory Method e Prototype nel caso in cui le classi degli oggetti da creare siano enne. Possiamo dire che l'Abstract Factory è un pattern composto che usa al proprio interno questi altri patterns.

Builder

Serve per **separare l'algoritmo di costruzione di una struttura di oggetti dalla creazione dei singoli oggetti**.

Una struttura di oggetti è formata da vari oggetti legati tra loro tramite riferimenti. Quando si crea la struttura occorre assicurarsi che tutte le sue parti vengano create. Invece di distribuire la responsabilità della creazione tra i vari oggetti (una oggetto crea un altro oggetto che crea altri oggetti ecc.) è più conveniente localizzare in un unico punto la logica di creazione dell'intera struttura al fine di rendere più flessibile eventuali modifiche della struttura e del processo di creazione. I partecipanti al pattern del Builder sono:

- *Director* incapsula l'algoritmo e la logica di creazione della struttura di oggetti;
- *Builder* costruisce la struttura istanziando i vari oggetti sotto la guida del *Director*;
- *Composite* è l'oggetto composto ritornato dal processo di creazione (la struttura).

```
class Composite {}

class Builder {
      createPartA(...);
      createPartB(...);
      createPartC(...);
      Composite getComposite();
}

class Director {
      Builder builder;
      Director(Builder builder) {this.builder = builder;}

      Composite createComposite() {
            // here is the logic to build the Composite
            if (...) builder.createPartA(...);
            while (...) builder.createPartB(...);
            ...
            return builder.getComposite();
      }
}
```

Patterns di struttura

Questi patterns si occupano di come oggetti e classi sono composti per formare strutture complesse.

Facade

Serve a **fornire un'unica interfaccia di accesso a un sistema complesso evitando le dipendenze sulla struttura interna del sistema**.

Una tecnica comune per definire un design complesso è quella dei **Layers** [GOF2], in cui un sistema complesso viene stratificato in diversi livelli corrispondenti a diversi livelli di astrazione. Si parte dal livello zero che lavora con gli elementi di livello più basso, fino al livello enne che fornisce le funzionalità di più alto livello. Si pensi ad esempio a un sistema operativo. Il livello zero è il kernel che gestisce direttamente le risorse hardware. Il livello uno usa il kernel per fornire funzionalità più astratte, come l'allocazione di blocchi logici su disco astraendo dall'allocazione fisica. Infine, il livello finale è quello offerto direttamente all'utente che vede solo i comandi di *copy*, *remove*, ecc. Il passaggio da un livello al successivo deve essere disciplinato in modo da ridurre le dipendenze tra i livelli e permettere un'evoluzione indipendente. Per ottenere ciò ogni livello definisce un'interfaccia, detta *Facade*, nei confronti del livello superiore che maschera la struttura degli oggetti del livello e fornisce un unico punto di accesso. È come se un intero sistema di oggetti venisse incapsulato in un unico oggetto che fornisce l'interfaccia del *Facade*.

```
// Layer 1
class Object1Layer1 {}
class Object2Layer1 {}
...
class ObjectNLayer1 {}

interface FacadeLayer1 {
      public void method() {
            // use ObjectNLayer1
      }
}

// Layer 2
class ObjectNLayer2 {
      facadeLayer1.method();
}
```

Composite (Container)

Serve a **rappresentare un oggetto complesso in una struttura con relazione di contenimento**.

Il tipico uso di questo pattern è all'interno di frameworks grafici come AWT e Swing. Gli elementi grafici che formano un'interfaccia sono organizzati secondo una gerarchia di contenimento: una finestra contiene dei pannelli che contengono dei bottoni, ecc. Nei casi in cui la relazione gerarchica di contenimento sia pervasiva nel framework è opportuno astrarre le proprietà comuni della relazione contenitore/componente all'interno di super classi specializzate. I partecipanti al pattern del Composite sono:

- *Component* è una classe astratta che implementa il comportamento di default degli oggetti nella gerarchia;
- *Container* è una classe astratta che estende *Component* aggiungendo il comportamento necessario per gestire le sotto componenti;
- *ConcreteComponent1,..., ConcreteComponentN* estendono *Component* e sono le classi che realizzano le componenti;
- *ConcreteContainer1,..., ConcreteContainerN* estendono *Container* e sono le classi che realizzano i contenitori;

Esempi di *ConcreteComponent* sono *Button*, *CheckBox*, ecc. Esempi di *ConcreteContainer* sono *Panel*, *ScrollPane*, ecc. Quando un *Component* riceve una un'invocazione di metodo esegue l'operazione richiesta e, nel caso il *Component* sia in realtà un *Container*, la inoltra alle sue componenti e così via in modo ricorsivo (es. una richiesta di `paint`).

```
abstract class Component {
      // common state for all components
      Container parent;
      ...
      // common methods for all components
      ...
}

abstract class Container extends Component {
      // common state for all containers
      Component components[];
      ...
      // common methods for all containers
      add(Component c) {...}
      remove(Component c) {...}
      ...
}
```

Un aspetto importante di questo pattern è cercare di inserire all'interno di *Component* e *Container* il maggior numero di funzionalità. In questo modo le sottoclassi *ConcreteComponentN* e *ConcreteContainerN* devono aggiungere o modificare solo una parte specializzata e ridotta di comportamento.

Adapter

Permette a **una classe di supportare un'interfaccia anche quando la classe non implementa direttamente quell'interfaccia**.

Una classe Java supporta un'interfaccia quando implementa l'interfaccia, ossia implementa i metodi dichiarati dall'interfaccia.

```
interface AnInterface {
      public void aMethod();
}
class AClass implements AnInterface {
      public void aMethod() {...}
}
```

Ovviamente per fare ciò la classe deve essere costruita avendo in mente l'interfaccia, o le interfacce, che deve implementare. Ma esistono casi in cui questo non è possibile:

- La classe è già definita e non può essere modificata, es. una classe di libreria o di framework. Ad esempio, nei frameworks grafici come AWT e Swing, esistono oggetti sensibili alle azioni utente (bottoni, menu, ecc.). Ovviamente questi oggetti grafici non trattano esplicitamente le richieste: è l'applicazione, o l'applet, che deve definire il comportamento da associare a ogni azione;
- La classe deve implementare due o più interfacce diverse che hanno dei metodi con lo stesso nome e parametri ma semantica diversa.

Per risolvere il problema si definisce una classe intermedia, detta *Adapter*, che serve ad accoppiare l'interfaccia con la classe facendo da tramite tra le due. I partecipanti al pattern dell'Adapter sono:

- *Target* è l'interfaccia da implementare;
- *Adaptee* è la classe che fornisce l'implementazione;
- *Adapter* è la classe intermedia che implementa *Target* e richiama *Adaptee*.

Esistono due varianti dell'Adapter, una sfrutta l'ereditarietà e l'altra la delegazione. Nel primo caso *Adapter* estende *Adaptee*, mentre nel secondo contiene il riferimento a un oggetto di tipo *Adaptee* e ne delega il comportamento:

```
interface Target {
      public void aMethod();
}

class Adaptee {
      public void aDifferentMethod() {...}
}

class InheritanceAdapter extends Adaptee implements Target {
      public void aMethod() {
            aDifferentMethod();
      }
}

class DelegationAdapter implements Target {
      private Adaptee adaptee;
      DelegationAdapter(Adaptee a) {adaptee = a;}

      public void aMethod() {
            adaptee.aDifferentMethod();
      }
}
```

L'approccio basato su ereditarietà ha queste caratteristiche:
- siccome estende *Adaptee* non può funzionare per oggetti che sono istanze di sottoclassi di *Adaptee*;
- permette di modificare parte del comportamento dei metodi di *Adaptee*, con vantaggi e svantaggi conseguenti;
- non introduce nuovi oggetti e riferimenti, per cui il design è più pulito e facile da capire.

L'approccio basato su delegazione ha queste caratteristiche:
- permette a un'unica classe *Adapter* di lavorare con *Adaptee* e tutte le sue sottoclassi;
- introduce un oggetto ulteriore complicando il design;

Un uso tipico del pattern dell'Adapter in Java si ha con l'uso dei frameworks come AWT e Swing nei quali gli oggetti grafici sensibili alle azioni utente (bottoni, menu, ecc.) non trattano esplicitamente le richieste perché è l'applicazione, o l'applet, che deve definire il comportamento da associare a ogni azione. Gli oggetti grafici usano l'idioma del **Listener** per notificare l'occorrenza di una certa azione e l'applicazione o applet deve implementare i metodi dell'interfaccia *Listener*. Un idioma Java particolare in questi casi è l'uso di *inner-classes* anonime come Adapters tra l'interfaccia (Target) che riceve le azioni grafiche e l'applicazione o applet (Adaptee) che ne fornisce il comportamento:

```
// the Target interface
interface ActionListener {
      public void actionPerformed(ActionEvent e);
}

// the Adaptee (applet)
class Adaptee extends Applet {
      Button aButton = new Button("Press me");
      MenuItem anItem = new MenuItem("Open File");
      ...
      // the Button Adapter
      aButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                  buttonPressed(e);
            }
      });

      // the MenuItem Adapter
      anItem.addActionListener(new ActionListener() {
```

```
            public void actionPerformed(ActionEvent e) {
                  if (e.getActionName().equals("Open File"))
                        receiver.openFile(e);
                  else if....
            }
      });
      buttonPressed(ActionEvent e) {...}
      openFile(ActionEvent e) {...}
}
```

Proxy (Stub)

Serve quando si vuole **introdurre un surrogato, o rappresentante, di un oggetto**.

Uno dei motivi per introdurre un surrogato è quello di mantenere lo stesso livello di astrazione che si avrebbe nell'accedere direttamente all'oggetto mascherando complessità aggiuntive. Ad esempio, in Java RMI il client di un oggetto remoto non deve preoccuparsi delle problematiche inerenti alla comunicazione (protocolli, marshaling dei parametri, ecc.). Il client accede all'oggetto remoto come se fosse un oggetto locale. Per fare questo Java RMI introduce lato client un oggetto *Proxy*, detto **Stub** nella terminologia RMI, che fornisce la stessa interfaccia del server e incapsula la complessità della comunicazione remota. Un altro motivo è dare una rappresentazione object-oriented a elementi che non sono oggetti. Ad esempio, un sistema a oggetti che si interfaccia a un database relazionale può incapsulare la logica di accesso al database all'interno di oggetti *Proxy*. In questo modo il client rimane mascherato dell'esistenza del database e dalla sua complessità di accesso. I partecipanti al pattern del Proxy sono:

- *Target* è l'interfaccia dell'oggetto server remoto;
- *Server* è la classe che implementa *Target*
- *Proxy* fornisce la stessa interfaccia *Target* e incapsula la complessità per accedere a *Server*.

```
interface Target extends Remote {
      public void aMethod();
}

class Server extends UnicastRemoteObject implements Target {
      public void aMethod() {...}
}

// generated by the rmi compiler
class ServerStub extends RemoteStub implements Target {
      public void aMethod() {
            // forward the invocation to the remote object
            ...
      }
}
```

Decorator (Filter)

Permette di **aggiungere o modificare dinamicamente il comportamento di un oggetto**.

Il modo tradizionale nei linguaggi a oggetti per aggiungere o modificare il comportamento di una classe è tramite ereditarietà in cui si definisce una sottoclasse che introduce i cambiamenti richiesti. Talvolta occorre aggiungere o modificare il comportamento di un singolo oggetto e non di una classe perché si vuole che il nuovo comportamento venga introdotto a runtime. L'effetto si ottiene con il pattern del Decorator, anche se il nome **Filter** corrisponde alla nomenclatura Java del package `java.io` dove questo pattern viene ampiamente usato nella gestione degli streams e rende meglio l'idea del suo funzionamento. I partecipanti al pattern sono:

- *Target* è la classe dell'oggetto le cui funzionalità devono essere modificate a runtime;

- *Decorator* è un oggetto interposto davanti a *Target* che offre la stessa interfaccia e introduce il nuovo comportamento delegando poi a *Target* l'esecuzione normale.

```
interface Target {
      public void aMethod();
}

class Target extends Target {
      public void aMethod() {
            // some behavior
            ...
      }
}

class Decorator extends Target {
      private Target target;
      Decorator(Target t) { target = t;}

      public void aMethod() {
            // some additional behavior
            ...
            target.aMethod();
            // some additional behavior
            ...
      }
}
```

Due considerazioni:

1 *Target* può essere un'interfaccia o una classe. Nel primo caso il design è più pulito perché *Decorator* può implementare direttamente *Target*. Nel secondo caso *Decorator* deve estendere *Target* con lo svantaggio di duplicare lo stato dell'oggetto;

2 Il pattern del Decorator può essere ricorsivo, nel senso che è possibile mettere davanti a un *Decorator* un altro *Decorator* e un altro ancora, ecc. costruendo una catena di *Decorators*. L'effetto è di avare una granularità molto fine sulla modifica dinamica del comportamento.

Flyweight

Permette di **condividere oggetti qualora un loro numero elevato avrebbe costi proibitivi di efficienza e gestione**.

Un design a oggetti spinto all'estremo può portare alla definizione di grande numero di classi dove ogni entità del dominio viene rappresentata come un oggetto. In questi casi il numero di oggetti creati a runtime è elevato e il sistema potrebbe collassare sotto il peso della gestione della memoria. Ecco perché spesso occorre fare compromessi sulla granularità delle classi e numero di oggetti istanziati. Ad esempio, supponiamo di definire un word processor con un design a oggetti. Possiamo rappresentare tabelle e figure come oggetti, ma difficilmente ci spingeremmo a trattare i singoli caratteri come oggetti, anche se una rappresentazione a oggetti permetterebbe una flessibilità molto fine (estensione del set di caratteri, uniformità nel trattare la formattazione, ecc.). Il pattern del Flyweight risolve il problema.

Un *Flyweight* è un oggetto condiviso in contesti diversi e ogni contesto vede il *Flyweight* come un oggetto indipendente. Dal punto di vista logico è come se venissero creati oggetti diversi, mentre gli oggetti sono condivisi fisicamente. Nel paradigma object-oriented ogni oggetto, per definizione, incapsula uno stato proprio e oggetti diversi hanno stato diverso. Fare credere che uno stesso oggetto si comporti come due o più oggetti sembra contrario ai principi dell'object-orientation. Nel pattern del Flyweight il punto chiave è distinguere tra stato *indipendente* e stato *dipendente* dal contesto. Lo stato indipendente dal contesto è quello che non varia e può rimanere all'interno del *Flyweight*. Lo stato dipendente dal contesto deve essere "esternalizzato", ossia rimosso dal *Flyweight* e passato come parametro dei metodi. I partecipanti al pattern del Flyweight sono:

- *Flyweight* è l'oggetto condiviso. Incapsula lo stato indipendente dal contesto e riceve quello dipendente come parametro dei suoi metodi;
- *FlyweightFactory* crea e gestisce i *Flyweights*. I clients devono ottenere i riferimenti ai *Flyweight* tramite la *FlyweightFactory* e mai istanziare direttamente i *Flyweights*. La *FlyweightFactory* mantiene un dizionario (hashtable) e permette ai clients di individuare un *Flyweight* di interesse passando una chiave, ad esempio un nome.

```
class Flyweight {
      // independent state

      void aMethod(// dependent state) {...}
}

class FlyweightFactory {
      private Hashtable registry = new Hashtable();

      Flyweight getFlyweight(Object key) {
            Flyweight flyweight = (Flyweight)registry.get(key);
            if (flyweight == null) {
                  flyweight = new Flyweight();
                  registry.put(key, flyweight);
            }
            return flyweight;
      }
}
```

Patterns di comportamento

Questi patterns si occupano degli algoritmi e della distribuzione di responsabilità tra oggetti. Non dunque le relazioni tra oggetti e classi, ma il loro modo di comunicare e il flusso di controllo a runtime tra gli oggetti.

Template Method

Permette di **definire un algoritmo in una classe lasciando alle sottoclassi la possibilità di modificare alcune parti dell'algoritmo senza alterarne la struttura**.

Questo è in assoluto uno dei patterns più importanti nel design a oggetti perché costituisce il modo più pulito e controllato per il riuso del codice mediante ereditarietà. L'ereditarietà è lo strumento che permette alle sottoclassi di modificare il comportamento di una classe. La granularità della modifica è a livello di metodo: se si ridefinisce un metodo occorre riscrivere tutto il comportamento del metodo. Alle volte capita di voler definire un algoritmo all'interno di un metodo che deve rimanere inalterato nella sua struttura principale, e allo stesso tempo si vuole dare la possibilità alle sottoclassi di ridefinire alcuni passi dell'algoritmo. Ad esempio, il metodo di `openDocument` di una classe *Document* definisce ogni passo per aprire un documento:

```
void openDocument(String name) throws DocumentEception {
      if (!canOpen(name))
             throw new DocumentEception("cannot open " + name);
      Document doc = createDocument(name);
      if (doc != null) {
            openingDocument(doc);
            doc.open();
            documentOpened(doc);
      }
}
```

La struttura dell'algoritmo in `openDocument` non deve cambiare nelle sottoclassi di *Document*. Ciò che può cambiare sono i passi dell'algoritmo, ossia i metodi `canOpen`, `openingDocument`,

`documentOpened`, definiti come metodi astratti nella classe *Document* e la cui semantica varia a seconda che si tratti di un documento testo, disegno, ecc. e quindi della sottoclasse. Il metodo `openDocument` è un Template Method che fissa l'ordine e la sequenza delle operazioni richiamate, ma lascia alle sottoclassi la possibilità di ridefinire le operazioni stesse. I partecipanti al pattern del Template Method sono:

- *AbstractClass* implementa il metodo `templateMethod` che definisce la struttura dell'algoritmo e invoca altri metodi astratti come passi dell'algoritmo;
- *ConcreteClass* estende *AbstractClass* e implementa i metodi astratti, ossia i passi dell'algoritmo.

```
abstract class AbstractClass {
      private void templateMethod() {
            // here's the algoritm
            ...
            method1();
            while (...) method2();
            ...
            method3();
      }
      abstract void method1();
      abstract void method2();
      abstract void method3();
}

class ConcreteClass etends AbstractClass {
      void method1() {...}
      void method2() {...}
      void method3() {...}
}
```

`templateMethod` è **privato** a indicare che lo sottoclassi non possono ridefinire l'algoritmo. Una variante del pattern del Template Method non dichiara i passi dell'algoritmo come metodi astratti bensì fornisce un comportamento di default a tali metodi, eventualmente nullo. È estremamente importante quando si definisce il `templateMethod` precisare quali passi dell'algoritmo devono essere implementati dalle sottoclassi, e in tal caso andranno dichiarati **astratti** e quali possono essere implementati, e in tal caso avranno un comportamento di default. Questo approccio prende il nome di *Hollywood Principle* con il motto "don't call us, we'll call you", ossia il fatto che è la super classe a richiamare i metodi della sottoclasse e non viceversa come succede applicando alla regola il concetto dell'ereditarietà.

```
abstract class AbstractClass {
      abtract void method1();       // must be redefined
      void method2() {}             // null behaviour: can be redefined
      ...
}
```

Chain of Responsibility

Serve per avere **una catena di oggetti dove ogni oggetto può decidere se eseguire l'operazione richiesta o delegarla all'oggetto successivo**.

Il pattern Chain of Responsibility è la generalizzazione del concetto di delegazione dei linguaggi a oggetti. Spesso due o più oggetti hanno una relazione di delegazione dove un oggetto, il delegante, delega la responsabilità per l'esecuzione di una certa funzionalità a un secondo oggetto, il delegato. Se il delegato delega a sua volta l'esecuzione della funzionalità a un terzo oggetto otteniamo una catena di delegazione delle responsabilità. I partecipanti al pattern Chain of Responsibility sono:

- *Handler* è l'interfaccia comune di tutte i delegati;

- *ConcreteHandler1,..., ConcreteHandlerN* implementano *Handler* e mantengono il riferimento al prossimo delegato.

```
interface Handler {
      public void aMethod();
      ...
}

class ConcreteHandler1 implements Handler {
      private Handler next;
      ...
      public void aMethod() {
            if (...) {
                  // handle request
                  return;
            } else next.aMethod();
      }
}
```

Iterator (Enumeration)

Fornisce un modo per **accedere agli elementi di una collezione di oggetti in modo sequenziale senza esporre la struttura interna della collezione**.

Una collezione di oggetti, come una lista, un vettore o un dizionario (hashtable), deve dare modo di percorrere i suoi elementi uno dopo l'altro, magari per eseguire una certa operazione su tutti. L'idea del pattern dell'Iterator è di estrarre la responsabilità per l'attraversamento della collezione all'interno di un oggetto iteratore. In tal modo si possono definire iteratori con diverse politiche di attraversamento senza doverle prevedere all'interno della collezione. I partecipanti al pattern sono:

- *Aggregate* definisce l'interfaccia per l'accesso alla collezione;
- *ConcreteAggregate* è la classe che implementa *Aggregate* e contiene la struttura della collezione;
- *Iterator* definisce l'interfaccia dell'iteratore;
- *ConcreteIterator* implementa *Iterator* e mantiene il riferimento all'elemento corrente nella collezione.

Ogni collezione è responsabile della creazione dei suoi iteratori. Quindi l'interfaccia *Aggregate* deve definire un metodo `createIterator` implementato dalla classe *ConcreteIterator* usando il pattern del **Factory Method**.

```
interface Aggregate {
      public Iterator createIterator();
}

interface Iterator {
      Object getFirstElement();
      Object getCurrentElement();
      Object getNextElement();
      boolean hasMoreElements();
}

// the concrete iterator
class ListIterator implements Iterator {
      private int current = 0;
      private List list;
      ListIterator(List l) {list = l;}

      Object getNextElement() {
            list.getElementAt(current++);
```

```
      }
      boolean hasMoreElement() {
            return (current < list.getSize());
      }
      ...
}

// the concrete aggregate
class List implements Aggregate {
      private Object[] elements;

      Iterator createIterator() {
            return (Iterator)new ListIterator(this);
      }
}
```

In Java questo pattern è presente con l'idioma **Enumeration** nelle collezioni *Vector* e *Hashtable*:

```
interface Enumeration {
      Object getNextElement();
      boolean hasMoreElements();
}

// the concrete aggregate
class Vector {
      private Object[] elements;

      Enumeration elements() {
            return (Enumeration)new VectorEnumeration(this);
      }
}
```

Un uso tipico dell'idioma è il seguente, notate il downcasting dato che le collezioni sono fatte per oggetti generici di classe *Object*:

```
for (Enumeration e = vector.elements(); e.hasMoreElements();) {
      AnObject obj = (AnObject)e.nextElement();
      obj.aMethod();
}
```

È buona norma definire le collezioni separando interfaccia e implementazione. In Java 1.1 le collezioni *Vector* e *Hastable* non fanno questa distinzione, ma in Java 1.2 il supporto alle collezioni è stato raffinato e ora esiste una distinzione chiara tra cosa è la collezione (interfaccia) e come è realizzata (implementazione). Ad esempio, un dizionario (detto *Map*) ha tre implementazioni: come hashtable, adatto nei casi generali; come array, utile se il dizionario è di ridotte dimensioni e garantisce prestazioni migliori su operazioni di creazione e iterazioni; come albero bilanciato, che impone un ordinamento agli elementi e offre prestazioni migliori per le operazioni di ricerca ma è più costoso per le inserzioni di elementi. L'interfaccia *Enumeration* viene sostituita in Java 1.2 dalle interfacce *Iterator* e *ListIterator* che permetto un controllo molto più fine sugli oggetti della collezione mentre è in corso un'iterazione.

```
interface Iterator {
      public boolean hasNext();
      public Object next();
      public void remove();
}

interface ListIterator extends Iterator {
      public int nextIndex();
```

```
      public boolean hasPrevious();
      public Object previous();
      public int previousIndex();
      public void set(Object o);
}
```

Command

Serve a **trattare le richieste di operazioni come oggetti eliminando la dipendenza tra l'oggetto che invoca l'operazione e l'oggetto che ha la conoscenza di come eseguirla e permettere gestioni sofisticate di accodamento, sincronizzazione, gestione delle priorità, ecc.**

Alle volte è necessario inoltrare una richiesta a un oggetto senza conoscere il tipo di richiesta e l'oggetto che la deve trattare. Un esempio è il caso delle invocazioni a oggetti remoti in Java RMI. La componente che gestisce il protocollo di comunicazione non tratta direttamente la richiesta bensì la inoltra all'oggetto destinatario (dispatching). Un altro esempio è il caso in cui si vogliono disciplinare le invocazioni concorrenti a un oggetto in modo da gestire in modo sofisticato la coda di priorità delle richieste. Un oggetto *Scheduler* può decidere l'ordine di esecuzione e la politica di conflitto delle richieste. Il pattern del Command risolve il problema trasformando un'invocazione in un oggetto per permettere una gestione sofisticata di accodamento e esecuzione delle richieste. I partecipanti al pattern del Command sono:

- *Command* definisce un'interfaccia per eseguire un'azione `execute`;
- *ConcreteCommnad* implementa *Command* e definisce il comportamento metodo `execute`;
- *Invoker* crea un'istanza di *ConcreteCommand* e la passa allo *Scheduler* chiamando `manageCommand`;
- *Scheduler* è un'interfaccia che definisce il metodo `manageCommand`;
- *ConcreteScheduler* implementa `manageCommand` e gestisce le istanze di *ConcreteCommand* secondo una certa politica. Quando un *Command* deve essere eseguito lo *Scheduler* chiama il metodo `execute`.

L'*Invoker* crea un'istanza di *ConcreteCommand* e lo passa allo *Scheduler*. Lo *Scheduler* accoda il comando e in un momento opportuno chiama il metodo `execute`.

```
interface Command {
      public void execute();
}
class SetupConnectionCommand implements Command {
      private ConnectionManager manager;
      ButtonCommand(ConnectionManager m) {manager = m;}
      public void execute() {
            manager.connect();
      }
}

interface Scheduler {
      public void manageCommand(Command c);
}
// the ConcreteScheduler forks a thread pool to deal with the requests
class ConcreteScheduler implements Scheduler {
      ConcreteScheduler(int poolSize) {
            for (int i = 0; i < poolSize; ++i) new Thread(this).start();
      }
      public synchronized void manageCommand(Command c) {
            queue(c);
            notify();
      }
      public void run() {
            while (true) {
                  Command c;
```

```
                  synchronized (this) {
                        while (emptyQueue()) wait();
                        c = unqueue();
                  }
                  c.execute();
            }
      }
}

class Invoker {
      ...
      Command c = new OpenConnectionCommand(connectionManager);
      Scheduler.manageCommand(c);
      ...
```

Mediator

Serve a **ridurre i riferimenti espliciti tra oggetti incapsulando le loro regole di interazione in un oggetto separato**.

Un pattern con l'obiettivo di concentrare il comportamento invece di distribuirlo sembra andare contro i principi dell'object-orientation che invece incoraggia la distribuzione del comportamento e delle responsabilità tra gli oggetti. Sebbene partizionare un sistema in diversi oggetti aumenti la riusabilità, una complessa ragnatela di interconnessioni tra oggetti finisce per ridurla perché le dipendenze possono essere talmente complesse da rendere difficile ogni forma di evoluzione e modifica. Diventa problematico modificare il comportamento di un sistema quando la modifica di un oggetto può richiedere la modifica degli oggetti interconnessi a questo e così via. Nel pattern del Mediator esiste un oggetto mediatore che evita i riferimenti espliciti tra gli oggetti e controlla e coordina la loro interazione. Siccome ogni oggetto dialoga solo con il mediatore è possibile modificare il comportamento di un oggetto senza che questo abbia impatti sugli altri. Un uso tipico di questo pattern si ha quando si definisce un'applicazione, o applet, che aggrega diverse componenti ciascuna responsabile di un certo servizio. Invece di mettere in comunicazione diretta le componenti, l'applicazione può agire da mediatore riducendo la dipendenza reciproca delle componenti. I partecipanti al pattern del Mediator sono:

- *Mediator* definisce un'interfaccia per permettere la comunicazione tra i vari oggetti. Gli oggetti usano questa interfaccia per mandare e ricevere richieste;
- *ConcreteMediator* implementa *Mediator* e incapsula il comportamento di coordinamento tra gli oggetti.

```
interface Mediator {
      public receivedCoordinates(Avatar a, Coordinates c);
}

// the broker component
class Broker {
      Mediator mediator;
      Broker(Mediator m) {mediator = m;}
      void sendCoordinates(Coordinates c) {...}
      ...
      mediator.receivedCoordinates(avatar, coordinates);
}

// the vrml component
class VRML {
      Mediator mediator;
      Broker(Mediator m) {mediator = m;}
      void setAvatarPosition(Avatar a, Coordinates c) {...}
      Coordinates getAvatarPosition(Avatar a) {...}
}
```

```
// the mediator applet
class MyApplet extends Applet implements Mediator {
      Broker broker = new Broker(this);
      VRML vrml = new VRML(this);

      public void init() {
            for(;;) {
            Coordinates c = vrml.getAvatarPosition();
            if (c.hasChanged()) broker.sendCoordinates(c);
            ...
            }
      }
      public void receivedCoordinates(Avatar a, Coordinates c) {
            if (avatar == ....) {
                  ...
            vrml.setCoordinates(a, c);
      }
}
```

Observer

Definisce **una relazione tra un oggetto e un insieme di oggetti interessati a essere notificati quando il primo oggetto cambia di stato**.

Quando si partiziona un sistema in un insieme di oggetti cooperanti è importante mantenere gli oggetti consistenti tra loro quando cambiano di stato. Spesso l'oggetto che cambia di stato deve essere indipendente dagli oggetti interessati al cambiamento permettendo che il loro numero e identità possa variare dinamicamente. Mantenere riferimenti espliciti agli oggetti interessati produce un design complesso e poco flessibile. Il pattern del **Mediator** propone una soluzione centralizzata al problema. Un modo per affrontare il problema in modo distribuito anziché centralizzato è proposto dal pattern dell'Observer. I partecipanti al pattern sono:

- *Observer* è l'interfaccia che specifica il tipo di notifiche di eventi di interesse;
- *ConcreteObserver* implementa *Observer* e specifica il comportamento dei metodi in risposta agli eventi;
- *Subject* è la classe che cambia di stato e avverte gli *Observers* quando tali cambiamenti occorrono. Deve offrire due metodi per rispettivamente aggiungere e rimuovere gli *Observers* dalla lista degli interessati.

Un *Subject* ha due modi per comunicare i cambiamenti agli *Observers*. Nel primo modo il *Subject* passa come parametro di notifica tutta l'informazione necessaria per capire cosa è cambiato. Nel secondo modo il *Subject* non passa informazione, o passa informazione incompleta, e gli *Observers* interrogano il *Subject* se necessitano di informazione aggiuntiva. I due approcci hanno vantaggi e svantaggi. Il primo rischia di passare troppa informazione ma evita una comunicazione incrociata che potrebbe causare problemi di sincronizzazione (metodi **synchronized** in Java). Il secondo approccio ha vantaggi e svantaggi opposti.

```
interface Observer {
      public void update();
}

class ConcreteObserver implements Observer {
      ...
}

class Subject {
      public void addObserver(Observer o) {...}
      public void removeObserver(Observer o) {...}
      ...
      for (Enumeration e = observers.elements(); e.hasMoreElements();) {
```

```
            Observer o (Observer)e.nextElement();
            o.update();
      }
}
```

Il pattern dell'Observer è sicuramente uno dei più usati in Java con l'idioma del **Listener**, fondamentale nella descrizione del modello a componenti Java Beans e nei frameworks grafici AWT e Swing. Uno dei requisiti fondamentali di un modello a componenti è mantenere un accoppiamento lasco tra gli oggetti in modo da permettere il facile *plug & play* delle componenti. L'idioma del Listener introduce una serie di convenzioni sui nomi al fine di rendere più semplice la comprensione e facilitare l'introspezione con Java Reflection. Le regole sono:

- Definire una sottoclasse di *java.util.EventObject* per ogni categoria di eventi generata dal *Subject*. Il nome della classe deve terminare con "Event", es. *MouseEvent*;
- Per ogni categoria di eventi definire un'interfaccia che estende *java.util.EventListener* il cui nome si ottiene sostituendo "Event" con "Listener", es. *MouseListener*. I metodi offerti corrispondono alle varie notifiche che il *Subject* vuole comunicare. Ogni metodo deve tornare `void` e accettare come solo parametro un oggetto di classe *MouseEvent*;
- Definire la classe del *Subject*. Per ogni categoria di eventi propagati da questa classe deve essere presente una coppia di metodi per permettere di aggiungere e rimuovere i *Listeners*. Il nome di questi metodi segue la convenzione `add<listener-interface>` e `remove<listener-interface>`, es:
  ```
  void addMouseListener(MouseListener listener);
  void removeMouseListener(MouseListener listener);
  ```
- Per ogni metodo nell'interfaccia *Listener* definire un metodo **privato** nel *Subject* con nome `fire<listener-method>` la cui implementazione crea un oggetto di tipo *MouseEvent* e propaga l'evento a tutti i *Listeners* registrati, es:
  ```
  private void fireMouseClicked();
  ```
  Definire un *ConcreteListener*, ossia una classe che implementa *MouseListener* e registra il proprio interesse verso il *Subject*.

```
// the event class
class ConferenceEvent extends java.util.EventObject {
      public ConferenceEvent(ConferenceManager source) {super(source);}
      ...
      public String getCallerAddress();
}

// the listener interface
interface ConferenceListener extends java.util.EventListener {
      public void conferenceRinging(ConferenceEvent e);
      public void conferenceEstablished(ConferenceEvent e);
      public void conferenceRejected(ConferenceEvent e);
}

// the subject class
class ConferenceManager {
  private Vector listeners = new Vector();

  private void fireConferenceRinging() {
      Vector temp;
      Synchronized (this) {
            temp = (Vector)listeners.clone();
      }
      ConferenceEvent event = new ConferenceEvent(this);
      for (Enumeration e = temp.elements(); e.hasMoreElements();) {
            ((ConferenceListener)e.nextElement()).conferenceRinging(event);
      }
  }
  public synchronized void addConferenceListener(ConferenceListener l) {
```

```
      listeners.addElement(l);
  }
  public synchronized void removeConferenceListener(ConferenceListener l) {
      listeners.removeElement(l);
  }
  ...
}

// the concrete listener
class MyApplet extends Applet implements ConferenceListener {
      ....
      manager.addConferenceListener(this);
      ...
      public void conferenceRinging(ConferenceEvent e) {
            // deal with the event
      }
}
```

Sebbene le regole definite sopra siano le più ortodosse dell'idioma (come da specifica Java Beans), è possibile rilassarle con queste varianti:

- si può evitare la classe *Event* lasciando che i metodi dell'interfaccia *Listener* possano accettare qualunque tipo e numero di parametri;
- il metodo privato `fire<listener-method>` è una convenzione ma non è obbligatorio;
- si può evitare la sincronizzazione dei metodi add/remove se l'applicazione non è mulit-threaded, così come la clonazione del *Vector* che serve solo per garantire la consistenza in caso di accessi concorrenti.

Ultima cosa prima di concludere. Accanto alle classi e interfacce sopra descritte, l'idioma prevede una classe opzionale dal nome *<event-type>Adapter*, es. *MouseAdapter* che implementa *MouseListener* e fornisce un'implementazione nulla a tutti i metodi dell'interfaccia. In questo modo si possono definire i *ConcreteListeners* come sottoclassi di *MouseAdapter* e implementare solo i metodi di interesse. Purtroppo il nome "Adapter" è in questo caso infelice perché non ha nulla a che vedere con il pattern dell'**Adapter**.

State (Automata)

Permette a **un oggetto di cambiare comportamento quando cambia il suo stato interno**.

Consideriamo il caso di un oggetto che riflette il comportamento di un automa a stati. Ad esempio un oggetto che rappresenta una connessione TCP può essere in diversi stati: *established*, *listening*, *closed*, ecc. In corrispondenza di ogni stato della connessione il comportamento sarà diverso in risposta a richieste quali `open`, `close`, `send`, ecc.
Un modo per implementare un oggetto come automa a stati è avere per ciascun metodo dell'oggetto un controllo esplicito su quale stato si trova l'oggetto e agire di conseguenza. Quindi i vari metodi contengono una serie di `if`, `else if`, o `switch` che spesso rendono complesso il codice e difficile da modificare. Se aggiungiamo un nuovo stato dell'automa occorre modificare tutti i metodi coinvolti introducendo un nuovo ramo condizionale:

```
class Connection {
      int state = 0;

      Handle open() {
            switch (state) {
            case 0:
                  // do something
                  state = 1;
            case 1:
                  // do something else
                  state = 4;
            ...
```

```
            case N:
            }
      }
      void send(byte[] data) {
            switch (state) {
            case 0:
                  // do something
            case 1:
                  // do something else
            ...
            case N:
            }
}
```

Il pattern dello State offre un modo per implementare un automa a stati mantenendo più aderenza ai principi dell'object-orientation. L'idea è di definire una classe diversa per ogni stato dell'automa contenente la logica da eseguire in corrispondenza di quello stato. La classe mantiene anche i riferimenti agli stati adiacenti corrispondenti a una transizione, proprio come si modellerebbe un automa a stati. Aggiungere un nuovo stato corrisponde a definire una nuova classe con impatti limitati sulle altre. I partecipanti al pattern sono:

- *State* è l'interfaccia che definisce le operazioni da eseguire;
- *ConcreteState1*,..., *ConcreteStateN* sono le classi che implementano *State* e corrispondono ai vari stati dell'automa;
- *Context* è la classe che maschera e gestisce i vari stati dell'automa. I clients agiscono su *Context* e non vedono gli oggetti *States*.

```
interface State {
      public Handle open();
      public void send(byte[] data);
      ...
}
// the context
class Connection {
      State state;
      Handle open() {state.open(this);}
      void send(byte[] data) {state.send(this, data);}
      ...
}
// the concrete state
class ConnectionEstablished implements State {
      static ConnectionEstablished instance = new ConnectionEstablished();
      public Handle open(Connection c) {}
      public send(Connection c, byte[] data) {
            // send the data
      }
}
// the concrete state
class ConnectionClosed implements State {
      static ConnectionClosed instance = new ConnectionClosed();
      public Handle open(Connection c) {
            // open the connection
            ...
            // change the state
            c.state = ConnectionEstablished.getInstance();
      }
      public send(Connection c, byte[] data) {}
}
```

Notate che le classi corrispondenti agli stati dell'automa vengono spesso implementate come **Singleton**. Alternativamente possono essere implementate come **Fllyweight** lasciando al *Context* il

ruolo di *FlyweightFactory*. Nel caso in cui non sia possibile esternalizzare lo stato dipendente dal contesto degli oggetti (vedi Flyweight) allora occorre istanziare un oggetto *State* per ogni transizione, il che potrebbe risultare costoso.

Strategy

Permette di **incapsulare una famiglia di algoritmi sotto forma di oggetti intercambiabili**.

Per definizione un oggetto incapsula **stato** e **comportamento**. Il comportamento è dato dai metodi dell'oggetto che agiscono sullo stato. L'ereditarietà permette alle sottoclassi di ridefinire i metodi con algoritmi diversi. In alcuni casi potrebbe essere utile generalizzare il meccanismo per modificare il comportamento tanto da confinare comportamenti diversi in classi diverse. Il risultato è separare in classi diverse lo stato di un oggetto e il comportamento che agisce sullo stato permettendo grande flessibilità di combinazione tra i due, anche a runtime, cosa non possibile con l'ereditarietà. Non solo lo stato può essere accoppiato con algoritmi diversi, ma uno stesso algoritmo potrebbe essere applicato a diverse strutture dati. Un esempio di applicazione del pattern dello Strategy si ha in Java con l'idioma del **LayoutManager** nel framework AWT. Gli oggetti grafici che rappresentano dei contenitori (vedi il pattern del **Composite**) sono associati a dei *LayoutManager*, oggetti responsabili della disposizione delle componenti all'interno del contenitore, noto come algoritmo di "validazione" della gerarchia di contenimento. Estrarre la responsabilità di layout dal contenitore inserendola in un oggetto separato permette grande flessibilità. I partecipanti al pattern dello Strategy sono:

- *Strategy* è l'interfaccia comune alla famiglia di algoritmi;
- *ConcreteStrategy* implementa *Strategy* e contiene un algoritmo specifico;
- *Context* è la struttura dati su cui agisce un oggetto *Strategy*.

```
// the strategy
interface LayoutManager {
      void addLayoutComponent(String name, Component comp);
      void removeLayoutComponent(Component comp);
      ...
      // the strategy algorithm
      void layoutContainer(Container parent);
}

// the concrete strategies
class FlowLayout implements LayoutManager {
      ...
      void layoutContainer(Container parent) {...}
}
class BorderLayout implements LayoutManager {
      ...
      void layoutContainer(Container parent) {...}
}
class GridLayout implements LayoutManager {
      ...
      void layoutContainer(Container parent) {...}
}
...
Panel panel = new Panel();
panel.setLayout(new BorderLayout());
```

Analogamente al pattern dello **State**, le classi che implementano la famiglia di algoritmi si realizzano spesso come **Singleton** o **Flyweight**.

Visitor

Permette di **separare le classi di una struttura di elementi dalle operazioni applicate a queste**.

Capita spesso di organizzare oggetti in strutture dati complesse, come nel pattern del **Composite**. Consideriamo il caso in cui si vuole definire un'operazione su tutti gli elementi della struttura. Un modo è aggiungere l'operazione come metodo di tutte le classi coinvolte nella struttura. Questa soluzione è adatta se le operazioni da eseguire si possono prevedere nel momento in cui si progetta la struttura. Se per contro le operazioni applicabili sono decise in momenti successivi o possono variare con più facilità rispetto alla struttura, allora aggiungere e modificare ogni volta le classi della struttura produce un sistema difficile da capire, mantenere e modificare. Senza contare il fatto che magari le classi della struttura sono classi di libreria e quindi non possono essere modificate. Il pattern del Visitor permette di confinare le operazioni in classi separate rendendo le classi della struttura indipendenti da aggiunta e modifiche di operazioni. I partecipanti al pattern del Visitor sono:

- *Element* è un'interfaccia con l'operazione `accept` che riceve un *Visitor* come argomento;
- *ConcreteElementA,..., ConcreteElementZ* implementano *Element*. Il metodo `accept` deve essere implementato con la tecnica del **double-dispatch**;
- *Visitor* è un'interfaccia che definisce tante versioni dell'operazione `visit` (overloading) quanti sono le classi *ConcreteElementX*;
- *ConcreteVisitor1,..., ConcreteVisitorN* implementano *Visitor*. Ogni metodo `visit(ConcreteElementX)` implementa la versione dell'operazione da applicare alla classe *ConcreteElementX*.

```
interface Element {
      public void accept(Visitor visitor);
}

interface Visitor {
      public void visit(ConcreteElementA element);
      ...
      public void visit(ConcreteElementZ element);
}

class ConcreteElementA implements Element {
      public void accept(Visitor visitor) {visitor.visit(this);}
      // any other state and behaviour
      ...
}

class ConcreteVisitor1 implements Visitor {
      public void visit(ConcreteElementA element) {
            // perform operation 1 on element A
      }
      ...
}
```

L'interfaccia *Element* non è strettamente necessaria purché ogni classe *ConcreteElementX* implementi il metodo `accept`. Come si vede si hanno due gerarchie di classi parallele: le classi della struttura (*Elements*) e le classi delle operazioni applicate (*Visitors*). Aggiungere una nuova operazione corrisponde a creare una nuova classe *ConcreteVisitor*:

Sopra è stato detto che il metodo `accept` deve essere implementato con la tecnica del **double-dispatch**, vediamo di che si tratta. I linguaggi a oggetti come Java, C++ e Smalltalk supportano il meccanismo del **single-dispatch**, secondo il quale il metodo da eseguire dipende dal nome del metodo (con eventuale mangling dei parametri) e dal tipo dell'oggetto su cui viene invocato: il polimorfismo è essenzialmente questo. Il **double-dispatch** semplicemente significa che il metodo da eseguire dipende non da uno ma da due oggetti. Nei linguaggi sopra indicati il double-dispath si realizza con due applicazioni polimorfe del single-dispatch. Il metodo `accept` è implementato come:

```
visitor.visit(this);
```

e fa si che l'implementazione scelta del metodo `visit` dipenda da due classi: *ConcreteVisitorN* e *ConcreteElementX*. Infatti, prima viene scelta in modo polimorfo la classe *ConcreteElementX* nella

chiamata di `accept`, poi il metodo `accept` sceglie in modo polimorfo la classe *ConcreteVisitorN* nella chiamata di `visit`. Questo è l'elemento chiave del pattern del Visitor: l'operazione eseguita dipende sia dal tipo di operazione (*Visitor*) che dalla classe su cui viene applicata (*Element*).

Un elemento da sottolineare nel pattern del Visitor è che la sua convenienza dipende dalla stabilità della struttura. Se le classi che compongono la struttura sono relativamente stabili, mentre variano facilmente le operazioni applicate, allora il Visitor è la risposta. Se invece la struttura di classi può cambiare spesso, allora la convenienza del Visitor scompare perché aggiungere una classe *ConcreteElementY* significa modificare l'interfaccia *Visitor* e tutte le classi *ConcreteVisitor1*,..., *ConcreteVisitorN* aggiungendo un metodo `visit` specifico per *ConcreteElementY*.

Model/View/Controller (Model-UI)

Permette di **separare lo stato di un oggetto grafico dal modo in cui questo viene visualizzato e da come l'utente controlla l'interazione**.

Le applicazioni grafiche sono quelle più soggette ai cambiamenti. Portare l'applicazione su diverse piattaforme, Windows, Mac, Motif, ecc., richiede di adattare la grafica al Look & Feel del sistema. In altri casi su vorrebbe personalizzare l'applicazione per ogni cliente affinché questi possa decidere l'apparenza dell'interfaccia secondo le proprie preferenze. Se l'interfaccia grafica è altamente integrata con la logica di servizio dell'applicazione, ogni cambiamento all'interfaccia ha impatti sull'intera applicazione rendendo il processo di personalizzazione altamente costoso. Il pattern del Model/View/Controller (MVC) separa gli oggetti responsabili della visualizzazione grafica e controllo con l'utente dagli oggetti che contengono i dati e la logica dell'applicazione. Grazie a questa separazione è possibile progettare interfacce grafiche in cui la stessa informazione viene visualizzate in modi diversi. Ad esempio, le informazioni riguardanti la distribuzione delle vendite di un certo prodotto rispetto alle fasce di età potrebbe essere visualizzata contemporaneamente in una tabella, in un diagramma a torta e uno a barre. Se l'utente cambia alcuni dati nella tabella la modifica si riflette immediatamente sugli altri due diagrammi. Questo è possibile perché le tre viste fanno riferimento alla stessa informazione contenuta in un oggetto separato. I partecipanti del pattern MVC sono:

- *Model* è la classe che contiene la logica e i dati indipendenti da come sono visualizzati;
- *View* è la classe responsabile alla visualizzazione dei dati del *Model*. È possibile associare diverse *Views* allo stesso *Model*;
- *Controller* intercetta le azioni utente (es. click di mouse, pressione tasti, ecc.) e le propaga al *Model* e al *View*. Esiste un *Controller* per ogni *View*.

Il pattern MVC non è descritto nel catalogo [GOF1] perché considerato un pattern architetturale complesso. Credo comunque che la sua importanza nell'ambito delle interfacce grafiche sia tale da meritare una descrizione, anche perché l'architettura di Swing è basata su questo pattern. Che sia un pattern complesso è evidenziato dal fatto di usare diversi patterns più semplici al suo interno:

- L'accoppiamento lasco tra *Model* e *View/Controler* usa il pattern dell'**Observer**. *View* e *Controller* si registrano come *Observers* nei confronti del *Model* permettendo di avere diverse *Views* per lo stesso *Model*;
- La separazione *View/Controller* è un esempio del pattern **Strategy**. Una *View* può essere associata a diverse strategie in risposta alle azioni utente, ossia a diversi *Controllers*. Ad esempio, una *View* disabilitata (shadow) può essere associata ad un *Controller* che ignora ogni azione utente.

Swing definisce l'idioma **Model-UI** come implementazione specifica del pattern MVC con le seguenti varianti:

- le funzionalità di *View* e *Controller* sono supportate da un unico oggetto *Component*. Secondo i designers di Swing, la flessibilità ottenuta dalla separazione *View/Controller* non ripaga la complessità aggiuntiva;
- viene introdotto un oggetto *UI-delegate* che esternalizza con il pattern Strategy l'algoritmo di visualizzazione (rendering) al di fuori di *Component* permettendo il *puggable-look&feel* a runtime.

Siccome la struttura dell'idioma Model-UI è complessa, separerò la descrizione delle relazioni Model/Component e Component/UI-delegate.

Model/Component

I partecipanti alla relazione sono:

- *AbstractModel* è l'interfaccia che permette a *Component* di recuperare i dati da visualizzare;
- *ConcreteModel* implementa *AbstractModel* e contiene i dati da visualizzare;
- *ModelListener* è un'interfaccia per ricevere notifiche da *ConcreteModel*;
- *Component* implementa *ModelListener* e si registra presso il *ConcreteModel* per ricevere notifiche sulla variazione dei dati da visualizzare.

A differenza di AWT tutti le componenti grafiche Swing non mantengono i dati da visualizzare al loro interno ma si affidano a un model separato. Alcuni models servono solo a contenere lo stato della componete visuale, ad esempio se un bottone è abilitato, o quale elemento è selezionato in una lista, ecc. Altri models contengono dei dati propri dell'applicazione da visualizzare in qualche forma, ad esempio in formato lista, tabella, albero, ecc. Quando viene creata una componente grafica l'applicazione può indicare un *ConcreteModel* che fornisce i dati da visualizzare alla componente. Se l'applicazione non indica il *ConcreteModel* la componente crea un *ConcreteModel* di default. Swing attualmente definisce 11 interfacce *AbstractModel* a seconda delle componenti (es. *ButtonModel*, *ListModel*, *TreeMode*, ecc.). Difficilmente un'applicazione ha l'esigenza di cambiare il model di default di un *JButton* o una *JScrollBar*, ma può invece risultare utile definire un model specifico per *JTable*, *JTree* e altre componenti di visualizzazione complesse.

```
// the abstract model
interface ListModel {
      int getSize();
      Obejct getEleementAt(int index);
      addDataListener(DataListener l);
      removeDataListener(DataListener l);
}

// the concrete model
class DefaultListModel implements ListModel {
      private Vector elements = new Vector();
}

// the component
class JList extends JComponent implements Scrollable, Accessible {
      void setModel(ListModel model) {...}
      ListModel getModel() {...}
      ...
}
```

La comunicazione *Model/Component* usa l'idioma del **Listener** in cui il *Component* si registra come *Listener* del *ConcreteModel.* A seconda del tipo di *AbstractModel*, le notifiche possono trasportare informazione parziale o completa. Nel primo caso il *Component* consulta il *ConcreteModel* per recuperare i nuovi valori tutte le volte che viene notificata, mentre nel secondo caso la notifica contiene tutta l'informazione necessaria per capire la variazione. Ad esempio, *ButtonModel* utilizza l'approccio a informazione parziale, mentre *ListModel* e *TreeModel* quello a informazione completa.

Component/UI-delegate

I partecipanti alla relazione sono:

- *Component* è la componente grafica;
- *UI-delegate* è una classe astratta che definisce i metodi di visualizzazione richiamati da *Component*;
- *ConcreteUI-delegate* estende *UI-delegete* implementando un certo algoritmo di rendering;
- *ComponentListener* è un'interfaccia per ricevere notifiche da *Component.*

La possibilità di modificare il Look & Feel dell'interfaccia a runtime richiede un supporto del framework grafico non indirizzato direttamente dal pattern MVC. La separazione *View/Model* permette sì di modificare facilmente la parte grafica rispetto al resto dell'applicazione, ma richiede comunque un intervento a livello di programmazione perché occorre cambiare le classi che implementano le *Views* con altre classi.
Il concetto di *pluggable-look&feel* di Swing altro non è che l'uso del pattern **Strategy** per cambiare l'algoritmo di rendering a runtime, ossia il Look & Feel dell'interfaccia, separando la componente grafica *Component* dall'algoritmo di visualizzazione *UI-delegate*.

```
// the component
class JButton extends JComponent {
      JButton() {
            // create the ui-delegate
      }
      ButtonUI getUI() {...}
      void setUI(ButtonUI newUI) {...}
      String getUIClassID() {...}
      ...
}

// the abstract ui-delegate
abstract class ButtonUI extends ComponentUI {}
class BasicButtonUI extends ButtonUI {
      private static BasicButtonUI buttonUI = new BasicButtonUI();
      static BasicButtonUI createUI() {return buttonUI;}
      void installUI(JComponent c) {...}
      void uninstallUI(JComponent c) {...}
      void paint(Graphics g, JComponent c) {...}
      void update(Graphics g, JComponent c) {...}
      ...
}

// the concrete ui-delegate
class MetalButtonUI extends BasicButtonUI {...}
```

L'associazione *Component/UI-delegate* avviene quando la componente chiama il metodo `installUI` dello *UI-delagete*. Swing usa il pattern del **Flyweight** per creare gli *UI-delegates* delle componenti semplici (*JButton*, *JSlider*, ecc.) La classe *UI-delegate* crea un oggetto **Singleton** che non mantiene informazione di stato dipendente dal contesto. Ogni metodo di *UI-delegate* riceve come parametro l'oggetto *Component* particolare che può essere interrogato per avere le informazioni dipendenti dal contesto. Invece, per componenti più complesse non viene usato il pattern del Flyweight perché in questo caso l'esternalizzazione dello stato richiederebbe un alto numero di interazioni tra *Component* e *UI-delegate* vanificando i vantaggi del Flyweight. Questo testimonia come la scelta di un pattern sia sempre una sommatoria tra vantaggi e svantaggi. Per queste componenti complesse viene creata un'istanza *UI-delegate* per ogni componente.

```
class BasicTreeUI extends TreeUI {
      static BasicTreeUI createUI() {return new TreeUI();}
      ...
```

Quando lo *UI-delegate* viene associato a una componente, questi si registra come *ComponentListener* della componente per ricevere le notifiche di variazioni di stato da visualizzare.

Conclusioni

I lettori di [GOF1] si saranno accorti che dei 23 patterns descritti in quel libro io ne ho riportato 21. Ho voluto evitare il pattern del **Bridge** perché lo ritengo troppo simile, anche se cambia l'intento, al pattern dell'**Adapter**. Ho evitato il pattern del **Memento** perché a mio parere non è abbastanza

significativo. Ho invece aggiunto il pattern **Model/View/Controler** perché credo sia di importanza vitale come compendio di uso di altri patterns.

Allo stato attuale, l'approccio a patterns sembra essere l'arma più efficace per affrontare i problemi di design proprio per la difficoltà a introdurre regole certe in questo campo. Il loro successo deriva dal modo del tutto nuovo con cui si propongono: invece di tentare di mettere ordine in un mondo inerentemente caotico, meglio governare il disordine!

Riferimenti

[GOF1] E. Gamma, R. Helm, R. Johnson, J. Vlissides (Gang Of Four), “Design Patterns, Elements of Reusable Object-Oriented Software”, Addison Wesley, 1995.

[GOF2] F. Bushmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal (Gang OF Five), "A System of Patterns", John Wiley & Sons, 1996.