Design Patterns in Java

Dispensa per gli studenti del corso di Ingegneria del Software
Università di Camerino

A.A. 2010/2011

Contents

Introduzione

Questo lavoro descrive l'applicazione in Java delle soluzioni rappresentate nei 23 Design Patterns suggeriti da Gamma, Helm, Johnson e Vlissides (GoF)[1] nel libro "*Design Patterns: Elements of Reusable Object-Oriented Software*"[6], che diedero inizio allo studio e progettazioni di software riutilizzando soluzioni architetturali precedentemente testate. Il testo originale di questi autori è presentato, nella pubblicazione segnata, in formato di catalogo, e contiene esempi dell'uso di ogni singolo pattern in C++ e Smalltalk.

Il lavoro presentato in questo testo non corrisponde all'implementazione in Java dei Design Pattern, come si potrebbe inizialmente pensare. Si ha seguito un approccio leggermente diverso, consistente nello studio della motivazione di ogni singolo pattern e della soluzione architetturale proposta originalmente, per determinare come questa soluzione (se valida) può essere adeguatamente implementata in Java, sfruttando le API di base fornite con questo linguaggio.

Come risultato si ha avuto un importante gruppo di pattern la cui implementazione in Java non risulta strutturalmente diversa dalla proposta originale dei GoF. Tra questi, si trovano i pattern **Abstract Factory**, **Builder**, **Factory Method**, **Adapter**, **Bridge**, **Composite**, **Decorator**, **Façade**, **Flyweight**, **Mediator**, **Command**, **Interpreter**, **State**, **Strategy**, **Template Method**.

Al gruppo anteriore si dovrebbero aggiungere i pattern **Iterator** e **Observer**, che sono forniti come funzionalità delle Java API, il primo come strumento standard per la gestione delle collezioni, e il secondo come classi di base da estendere (con alcune limitazioni, come il fatto che la notifica agli *Observer* avvenga in un unico thread, come si discute nella sezione 19). Analogamente si possono mettere insieme ad essi il **Proxy** e il **Chain of Responsibility**, i quali possono essere implementati senza inconveniente alcuno, me che si ha voluto indicare separatamente perché Java gli utilizza come struttura di base per particolari servizi: il Proxy è la struttura sulla quale è supportata la Remote Method Invocation (RMI), intanto che una implementazione particolare della Chain of Responsibility è utilizzata per gestire le eccezioni, che vengono sollevate e portate via attraverso lo stack di chiamate di metodi, sino a trovare un metodo incaricato di gestirle[9].

Il **Prototype** è un pattern che deve essere analizzato separatamente, non solo perché Java fornisce importanti strumenti per creare copie di oggetti (clonazione e la serializzazione/deserializzazione), ma perché la soluzione del Prototype non risulta la più adeguata in Java, nei confronti di tutte le motivazioni che li diedero origine. Ad esempio, nel caso del caricamento di classi specificate in runtime (una delle giustificazioni del pattern), l'utilizzo degli strumenti presenti nella Reflection API possono fornire soluzioni più flessibili e poderose.

Altri due pattern, il **Memento** e il **Visitor**, sono stati implementati tramite una architettura leggermente diversa dalla suggerita da i GoF. In particolare, si ha sviluppato in questo lavoro una soluzione originale per il Memento, basata sull'uso di inner class, per gestire la esternalizzazione sicura dello stato di un oggetto (vedi sezione 18). In quanto riguarda il Visitor, nuovamente l'utilizzo della Reflection API conduce a una versione alternativa, più flessibile di quella concepita originariamente.

[1] GoF sta per "the Gang of Four", che corrisponde al nome con cui questi quattro autori sono conosciuti nel mondo dell'ingegneria del software.

Dentro l'insieme dei pattern il **Singleton** è quello che in Java si presta per una discussione più complessa, contrastante con la semplicità che c'è dietro la sua definizione. In linea di massima, si può dire che anche questo pattern può essere implementato in Java. Ci sono, però, diverse modalità di implementazione, secondo le caratteristiche che si vogliono tenere (funzionante in ambiente *single-thread*, oppure *multi-thread*, configurabile al momento dell'istanziazione, etc.), e perciò non tutte le possibilità sono sempre coperte. Il discorso di fondo non è che questo pattern sia più difficile di implementare in Java che in altri linguaggi, perché non ci sono eventualmente tutti gli strumenti necessari, ma tutt'altra cosa: dato che Java offre un insieme di funzionalità dentro le API di base (come, per esempio, l'invocazione remota di metodi, la serializzazione, ecc.) si aprono nuovi fronti che rendono più difficile progettare una implementazione in grado di garantire la funzionalità in tutti i casi.

Questo documento è organizzato secondo la stessa sequenza in cui i pattern sono stati presentati nel libro dei GoF. Ogni implementazione ha una descrizione strutturata nel seguente modo:

- Descrizione: una breve descrizione dell'obbiettivo del pattern, corrispondente in quasi tutti i casi al "intent" del libro di GoF.
- Esempio: si presenta un problema la cui soluzione si ottiene tramite l'applicazione del pattern.
- Descrizione della soluzione offerta dal pattern: si descrive testualmente l'architettura del pattern e come questa si applica al problema.
- Struttura del pattern: diagramma di classi in UML della struttura generica del pattern.
- Applicazione del pattern: offre un diagramma UML delle classi del problema, presenta l'abbinamento delle classi del problema con le classi che descrivono la struttura concettuale del pattern, descrive l'implementazione del il codice Java, e presenta e commenta gli output dell'esecuzione.
- Osservazioni sull'implementazione in Java: presenta gli aspetti particolari che riguardano l'implementazione del pattern in Java.

Le collaborazioni tra i pattern sono un aspetto molto importante da tenere in conto, perché molte volte portano all'implementazioni di soluzioni più adeguate nei confronti di un determinato problema. Nonostante ciò, in questo testo si ha deciso di presentare ogni singolo pattern separatamente, in modo che la comprensione di uno sia più semplice e non condizionata alla conoscenza di un altro. Pubblicazioni come quella dei GoF spiegano i modi in cui i pattern si possono fare interagire tra di loro.

Gli esempi presentati in ogni caso sono applicativi funzionanti che utilizzano i meccanismi più semplici di interazioni con l'utente (vale dire, tramite interfaccia di consola), in modo che la comprensione dell'esempio non risulti "disturbata" dalla presenza di codice necessario per la gestione dell'interfaccia grafica. A questo punto, si fa notare che anche per il Proxy pattern, tradizionalmente esemplificato in applicativi che gestiscono il caricamento di immagini, si ha sviluppato un esempio che non ha niente che vedere con interfaccia grafica.

La maggior parte degli esempi sono stati inventati appositamente per questo lavoro. Gli esempi presentati sono tutti diversi, in modo tale che ogni esempio possa essere capito singolarmente.

In quanto riguarda il materiale di supporto utilizzato, il testo di base fu la riferita monografia dei GoF. Altre fonti di informazioni importanti si trovarono sul Web. In particolare bisogna citare i diagrammi UML dei design pattern, sviluppati da Nikander[12], che furono utilizzati come base per la descrizione dei modelli, e diversi articoli di elevata qualità tecnica, riguardanti questo argomento, trovati sul sito *Javaworld*[2]. Altri siti visitati che offrono materiale di diversa natura e links verso altre

[2] http://www.javaworld.com/

pagine che trattano l'argomento dei degin patterns, sono *The Hillside Group*[3] e *Cetus Link Object Orientation*[4].

Come monografia complementaria al testo dei GoF, si ebbe a disposizione il libro di J. Cooper "Java Design Patterns: A tutorial"[4], il quale si dimostrò poco adeguato dal punto di vista didattico, per la in necessaria complessità degli esempi, per una mancata chiarezza nella descrizione dei pattern più complessi (e più interessanti da analizzare), e per limitarsi semplicemente a tradurre i modelli originali in Java, senza tenere conto degli strumenti che questo linguaggio offre.

Finalmente, si vuole indicare che Java è un linguaggio che consente l'applicazione di tutti i modelli proposti dai GoF, praticamente come una semplice "traduzione" della architettura originale. Se, però, si tengono in considerazione gli strumenti forniti col linguaggio, come la Reflection API, la serializzazione, o il Collections Framework, si possono progettare soluzioni ancora migliori per i problemi trattati dai pattern.

[3] http://hillside.net
[4] http://www.cetus-links.org/

Chapter 1

Abstract Factory

1.1 Descrizione

Presenta un'interfaccia per la creazione di famiglie di prodotti, in modo tale che il cliente che gli utilizza non abbia conoscenza delle loro concrete classi. Questo consente:

- Assicurarsi che il cliente crei soltanto prodotti vincolati fra di loro.
- L'utilizzo di diverse famiglie di prodotti da parte dello stesso cliente.

1.2 Esempio

Si pensi a un posto di vendita di sistemi Hi-Fi, dove si eseguono dimostrazioni dell'utilizzo di famiglie complete di prodotti. Specificamente si ipotizzi che esistono due famiglie di prodotti, basate su tecnologie diverse: una famiglia che ha come supporto il nastro (*tape*), e un'altra famiglia che ha come supporto il compact disc. In entrambi casi, ogni famiglia è composta dal supporto stesso (tape o cd), un masterizzatore (*recorder*) e un riproduttore (*player*).

Se si accetta il fatto che questi prodotti offrono agli utenti una stessa interfaccia (cosa non tanto distante dalla realtà), un cliente potrebbe essere in grado di eseguire lo stesso processo di prova su prodotti di entrambe famiglie di prodotti. Ad esempio, potrebbe eseguire prima una registrazione nel recorder, per poi dopo ascoltarla nel player.

Il problema consiste nella definizione di un modo di creare famiglie complete di prodotti, senza vincolare alla codifica del cliente che gli utilizza, il codice delle particolari famiglie.

1.3 Descrizione della soluzione offerta dal pattern

Il pattern "*Abstract Factory*" si basa sulla creazione di interfacce per ogni tipo di prodotto. Ci saranno poi concreti prodotti che implementano queste interfacce, stesse che consentiranno ai clienti di fare uso dei prodotti. Le famiglie di prodotti saranno create da un oggetto noto come *factory*. Ogni famiglia avrà una particolare *factory* che sarà utilizzata dal Cliente per creare le istanze dei prodotti. Siccome non si vuole legare al Cliente un tipo specifico di *factory* da utilizzare, le *factory* implementeranno una interfaccia comune che sarà dalla conoscenza del Cliente.

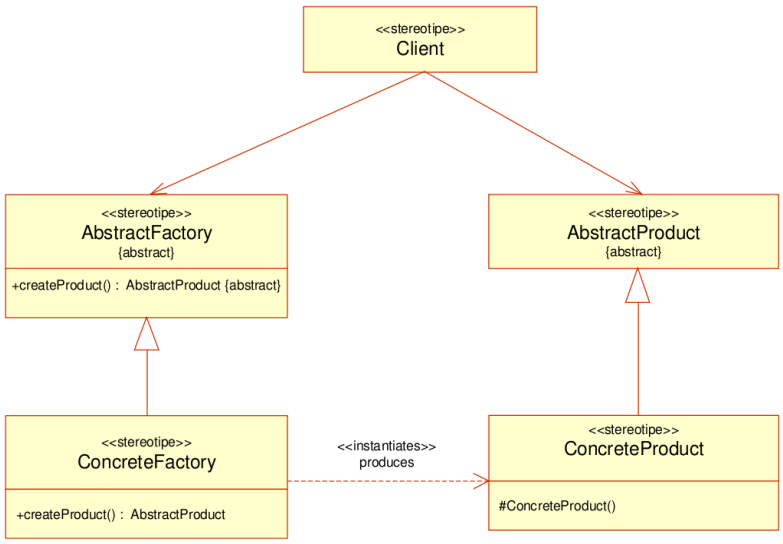


Figure 1.1: Struttura del Pattern

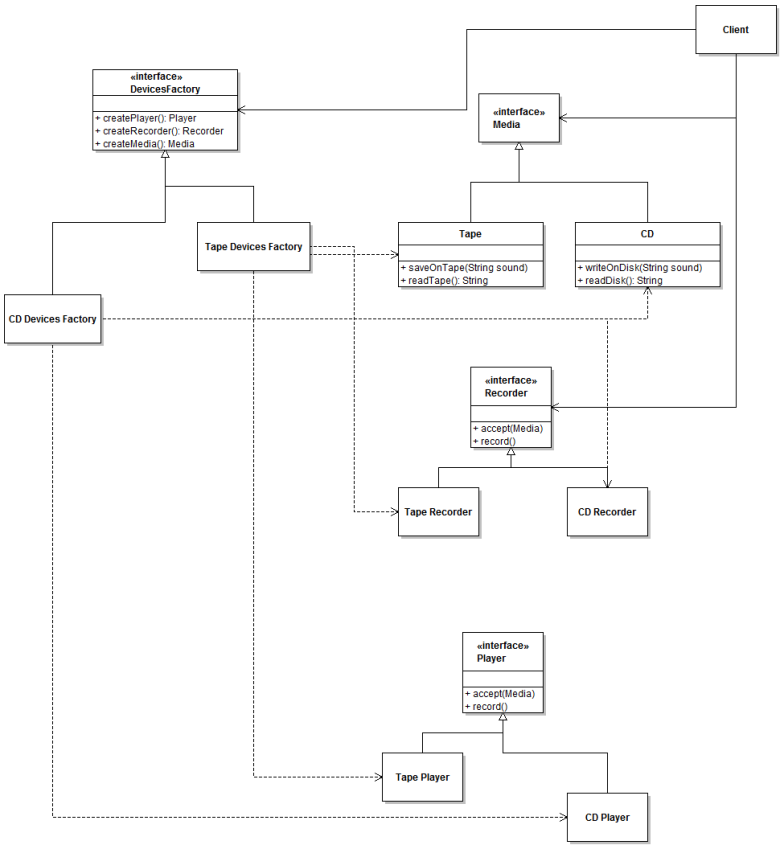


Figure 1.2: Schema del modello

1.4 Applicazione del Pattern

Partecipanti

- **AbstractFactory**: interfaccia DevicesFactory.
  - Dichiara una interfaccia per le operazioni che creano e restituiscono i prodotti.
  - Nella dichiarazione di ogni metodo, i prodotti restituiti sono dei tipi AbstractProduct.
- **ConcreteFactory**: classi TapeDevicesFactory e CDDevicesFactory.
  - Implementa l'AbstractFactory, fornendo le operazioni che creano e restituiscono oggetti corrispondenti a prodotti specifici (**ConcreteProduct**).
- **AbstractProduct**: interfacce Media, Recorder e Player.
  - Dichiarano le operazioni che caratterizzano i diversi tipi generici di prodotti.
- **ConcreteProduct**: classi Tape, TapeRecorder, TapePlayer, CD, CDRecorder e CDPlayer.
  - Definiscono i prodotti creati da ogni ConcreteFactory.
- **Client**: classe Client.
  - Utilizza l'**AbstractFactory** per rivolgersi alla **ConcreteFactory** di una famiglia di prodotti.
  - Utilizza i prodotti tramite la loro interfaccia **AbstractProduct**.

Descrizione del codice

Si creano le interfacce delle classi che devono implementare i prodotti di tutte le famiglie. *Media* una marker interface che serve per identificare i supporti di registrazione, intanto le interfacce *Player* e *Recorder* dichiarano i metodi che i clienti invocheranno nei riproduttori e registratori.

```
public interface Media { }

public interface Player {
        public void accept( Media med );
        public void play( );
}

public interface Recorder {
        public void accept( Media med );
        public void record( String sound );
}
```

S'implementano le concrete classi che costituiscono i prodotti delle famiglie. Le classi appartenenti alla famiglia di prodotti basati sul nastro sono *Tape*, *TapeRecorder* e *TapePlayer*:

```
public class Tape implements Media {
        private String tape= "";

        public void saveOnTape( String sound ) {
                tape = sound;
        }
```

```
        public String readTape( ) {
                return tape;
        }
}
```

```
public class TapeRecorder implements Recorder {
        Tape tapeInside;

        public void accept( Media med ) {
                tapeInside = (Tape) med;
        }

        public void record( String sound ) {
                if( tapeInside == null )
                        System.out.println( "Error: Insert a tape." );
                else
                        tapeInside.saveOnTape( sound );
        }
}
```

```
public class TapePlayer implements Player {
        Tape tapeInside;

        public void accept( Media med ) {
                tapeInside = (Tape) med;
        }

        public void play( ) {
                if( tapeInside == null )
                        System.out.println( "Error: Insert a tape." );
                else
                        System.out.println( tapeInside.readTape() );
        }
}
```

Le classi appartenenti alla famiglia di prodotti basati sul compact disc sono *CD*, *CDPlayer* e *CDRecorder*:

```
public class CD implements Media{
        private String track = "";

        public void writeOnDisk( String sound ) {
                track = sound;
        }

        public String readDisk( ) {
                return track;
        }
}
```

```
public class CDRecorder implements Recorder {
        CD cDInside;
```

```
        public void accept( Media med ) {
                cDInside = (CD) med;
        }

        public void record( String sound ) {
                if( cDInside == null )
                        System.out.println( "Error: No CD." );
                else
                        cDInside.writeOnDisk( sound );
        }
}
```

```
public class CDPlayer implements Player {
        CD cDInside;

        public void accept( Media med ) {
                cDInside = (CD) med;
        }

        public void play( ) {
                if( cDInside == null )
                        System.out.println( "Error: No CD." );
                else
                        System.out.println( cDInside.readDisk() );
        }
}
```

Linterfaccia *DevicesFactory* specifica la firma dei metodi che restituiscono gli oggetti di qualunque tipo di famiglia.

```
public interface DevicesFactory {
        public Player createPlayer();
        public Recorder createRecorder();
        public Media createMedia();
}
```

Si implementano le **ConcreteFactory**: *TapeDevicesFactory*, per la famiglia di prodotti basata sul nastro, e *CDDevicesFactory*, per quelli basati sul compact disc.

```
Public class TapeDevicesFactory implements DevicesFactory {
        public Player createPlayer() {
                return new TapePlayer();
        }

        public Recorder createRecorder() {
                return new TapeRecorder();
        }

        public Media createMedia() {
                return new Tape;
        }
}
```

```
public class CDDevicesFactory implements DevicesFactory {
```

```
        public Player createPlayer() {
                return new CDPlayer();
        }

        public Recorder createRecorder() {
                return new CDRecorder();
        }

        public Media createMedia() {
                return new CD();
        }
}
```

La classe *Client*Nome capitolo definisce gli oggetti che utilizzano i prodotti dogni famiglia. Il metodo *selectTechnology* riceve un oggetto corrispondente ad una famiglia particolare e lo registra dentro i propri attributi. Il metodo *test* crea una istanza dogni particolare tipo di prodotto e applica i diversi metodi forniti dalle interfacce che implementano i prodotti. Si deve notare che il cliente utilizza i prodotti, senza avere conoscenza di quali sono concretamente questi.

```
class Client {
        DevicesFactory technology;

        public void selectTechnology( DevicesFactory df ) {
                technology = df;
        }

        public void test(String song) {

                Media media = technology.createMedia();
                Recorder recorder = technology.createRecorder();
                Player player = technology.createPlayer();

                recorder.accept( media );
                System.out.println( "Recording the song : " + song );
                recorder.record( song );
                System.out.println( "Listening the record:" );
                player.accept( media );
                player.play();
        }
}
```

Finalmente, si presenta il programma che crea una istanza di un oggetto *Client*, il quale riceve tramite il metodo *selectTechnology* una istanza di una **ConcreteFactory**. In questo esempio particolare, si assegna una prova ad ogni famiglia di prodotti:

```
public class AbstractFactoryExample {
        public static void main ( String[] arg ) {

                Client client = new Client();

                System.out.println( **Testing tape devices );
                client.selectTechnology( new TapeDevicesFactory() );
                client.test( "I wanna hold your hand..." );

                System.out.println( **Testing CD devices );
```

```
C:\Patterns\Creational\Abstract Factory\>java AbstractFactoryExample

**Testing tape devices
Recording the song  : I wanna hold your hand...
Listening the record:
I wanna hold your hand...

**Testing CD devices
Recording the song  : Fly me to the moon...
Listening the record:
Fly me to the moon...
```

Figure 1.3: Esecuzione dell'esempio

```
                client.selectTechnology( new CDDevicesFactory() );
                client.test( "Fly me to the moon..." );
        }
}
```

Osservazioni sull'esempio

Nell'esempio si voluto accennare la stretta interazione tra le classi costituenti ogni famiglia di prodotto, in modo tale di costringere allutilizzo, nelle prove, di prodotti dello stesso tipo. In ogni particolare famiglia le interfacce tra i componenti sono diverse, e non conosciute dal cliente. In particolare, si noti che linterfaccia dogni tipo di Media verso i corrispondenti registratori e riproduttori varia da una famiglia allaltra.

1.5 Osservazioni sull'implementazione in Java

Dovuto al fatto che n l'**AbstractFactory** n gli **AbstractProduct** implementano operazioni, in Java diventa pi adeguato codificarli come interfacce piuttosto che come classi astratte.

Chapter 2

Builder

2.1 Descrizione

Separa la costruzione di un oggetto complesso dalla sua rappresentazione, in modo che lo stesso processo di costruzione consenta la creazione di diverse rappresentazioni.

2.2 Esempio

Per costruire modelli concettuali di dati, nellambito della progettazione di database, lo schema *Entity-Relationship* (ER) ampiamente utilizzato, anche se non esiste accordo su di ununica rappresentazione grafica di essi. Per esempio, per costruire un modello informativo nel quale si vuole indicare che *uno studente appartiene ad una singola universit, e ununiversit pu non avere studenti, oppure avere un numero illimitato di questi*, una delle notazioni diffuse consente di rappresentare questo fatto (figura 2.1).

Invece unaltra notazione rappresentata dalla figura 2.2.

Il primo schema, ad effetti di quest'esempio, verr chiamato modello non orientato, intanto che il secondo modello orientato[1].

Entrambi modelli sono praticamente equipollenti nei confronti di un progetto di sviluppo, essendo possibile notare che lunica differenza lomissione, nella seconda rappresentazione, del nome della relazione esistente tra le due entit[2], e il numero che indica la partecipazione minima dogni entit nella relazione.

Uno strumento di supporto alla costruzione di modelli, potrebbe non solo avere la possibilit di

[1] Questa una denominazione informale adoperata soltanto per fare un semplice riferimento ad uno o altro tipo di schema. Il secondo modello fu chiamato orientato per la presenza della freccia nella relazione, che determina linterpretazione della partecipazione (card.).

[2] E certo che se omesso il nome della relazione soltanto possibile rappresentare un unico vincolo tra due entit. Il modello presentato si semplificato soltanto per accennare differenze tra due prodotti a costruire, senza rendere pi complesso lesempio.

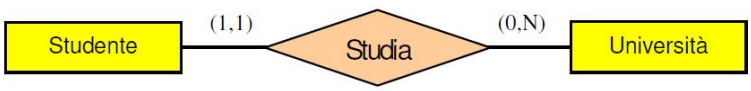


Figure 2.1: Builder ER 1

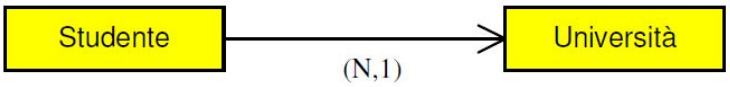


Figure 2.2: Builder ER 2

gestire entrambe tipologie di diagrammi, ma anche essere in grado di produrre i modelli in supporti (classi di oggetti) di diversa natura. Ad esempio, in un caso potrebbe essere richiesta la costruzione di un modello come un oggetto grafico, intanto che in un altro, la produzione di file di testo o di un documento XML che lo rappresenti. Nel caso di questesempio si deciso di avere un modello non orientato rappresentato come unistanza di una classe ERModel, e di un modello orientano rappresentato come una stringa.

Si osservi che siccome c una rappresentazione comune di base per entrambi modelli, il processo di costruzione in ogni caso sar diverso soltanto dal tipo di mattone utilizzato nella costruzione, piuttosto che nella logica del processo stesso. Per tanto, il problema consiste nella definizione di un sistema che consenta ottenere prodotti diversi, senza raddoppiare la logica del processo utilizzato.

2.3 Descrizione della soluzione offerta dal pattern

Il "*Builder*" pattern propone separare la "logica del processo di costruzione" dalla "costruzione stessa". Per fare ci si utilizza un oggetto Director, che determina la logica di costruzione del prodotto, e che invia le istruzioni necessarie ad un oggetto *Builder*, incaricato della sua realizzazione. Siccome i prodotti da realizzare sono di diversa natura, ci saranno *Builder* particolari per ogni tipo di prodotto, ma soltanto un unico *Director*, che nel processo di costruzione invocher i metodi del *Builder* scelto secondo il tipo di prodotto desiderato (i *Builder* dovranno implementare uninterfaccia comune per consentire al *Director* di interagire con tutti questi). Potrebbe capitare che per ottenere un prodotto particolare alcune tappe del processo di costruzione non debbano essere considerate da alcuni Builder (ad esempio, il Builder che costruisce i modelli non orientati, deve trascurare il nome della relazione e il grado della partecipazione minima).

2.4 Applicazione del Pattern

Partecipanti

- **Builder**: classe astratta ModelBuilder.
  - Dichiara una interfaccia per le operazioni che creano le parti dell'oggetto Product.
  - Implementa il comportamento default per ogni operazione.
- **ConcreteBuilder**: classi OrientedERBuilder e NotOrientedERBuilder.
  - Forniscono le operazioni concrete dell'interfaccia corrispondente al Builder.
  - Costruiscono e assemblano le parti del Product.
  - Forniscono un metodo per restituire il Product creato.
- **Director**: classe ERHardCodedDirector.
  - Costruisce il Product invocando i metodi dell'interfaccia del Builder.

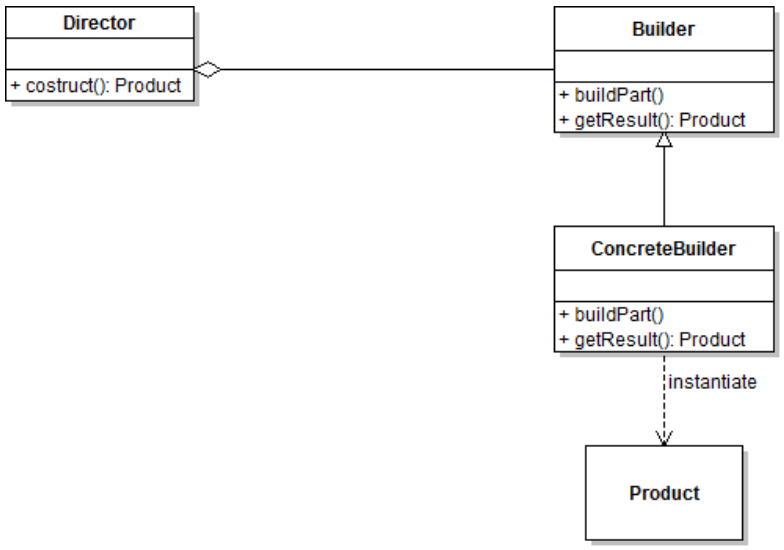


Figure 2.3: Struttura del Pattern

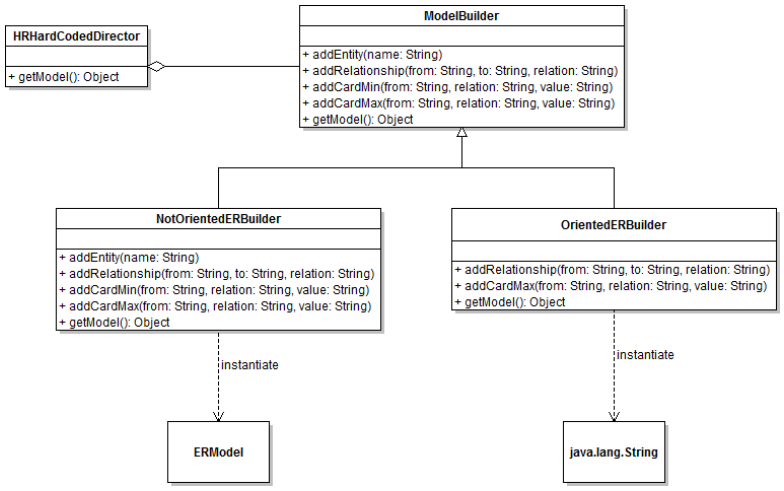


Figure 2.4: Schema del modello

- **Product**: classi String e ERModel.
  - Rappresenta l'oggetto complesso in costruzione. I ConcreteBuilders costruiscono la rappresentazione interna del Product.
  - Include classi che definiscono le parti costituenti del Product.

Descrizione del codice

Come è stato descritto, si è decisa la creazione di due rappresentazioni diverse, che sono contenute in oggetti di classe diversa:

- Un modello rappresentato in una lunga Stringa (String) contenente le primitive definite per lo schema non orientato. Un'istanza di questo sarà ad esempio:
  "[Student] - - - (N, 1) - - - ¿ [University]"
- Un modello rappresentato come un oggetto della classe ERModel, la cui definizione si presenta alla fine della descrizione dell'implementazione di questo pattern. Ad effetti dell'esemplificazione del *Builder pattern* si ipotizzi che questa classe è preesistente.

Si dichiara la classe astratta ModelBuilder dalla quale si specializzano i **ConcreteBuilders** per la costruzione di ogni tipologia di modello. La classe ModelBuilder include l'insieme di tutti i metodi che i **ConcreteBuilders** possiedono. I **ConcreteBuilders** implementano soltanto i metodi del loro interesse, per tanto il ModelBuilder li dichiara come metodi non astratti e fornisce codice di default per tutti questi (in questo caso codice nullo). Una eccezione è il metodo getModel dichiarato astratto nella classe ModelBuilder, che dovrà necessariamente essere implementato in ogni ConcreteBuilder con il codice necessario per restituire l'oggetto atteso come prodotto, una volta costruito. Dato che nel caso generale non si conosce la specifica tipologia dell'oggetto a restituire, il metodo getModel dichiara un Object come tipo di ritorno.

```
public abstract class ModelBuilder {
        public void addEntity( String name ) {};

        public void addRelationship(String fromEntity, String toEntity,
                String relation ) {};

        public void addCardMin( String entity, String relation, String value ) {};

        public void addCardMax( String entity, String relation, String value ) {};

        public abstract Object getModel();
}
```

In questo esempio, i metodi nei quali i **ConcreteBuilders** possono essereinteressati sono:

- addEntity: per aggiungere una entità.
- addRelationship: per aggiungere una relazione.
- addCardMin: per aggiungere partecipazione minima di una entità nella relazione.
- addCardMax: per aggiungere partecipazione massima di una entità nella relazione.
- getModel: per restituire l'oggetto costruito.

La classe NotOrientedERBuilder redefine tutti i metodi della superclasse ModelBuilder:

```
public class NotOrientedERBuilder extends ModelBuilder {
        private ERModel model;

        public NotOrientedERBuilder() {
                model = new ERModel();
        }

        public void addEntity( String name ) {
                model.addEntity( name );
        }

        public void addRelationship(String fromEntity, String toEntity,
                        String relation ) {
                model.addRelationship( fromEntity, toEntity, relation );
        }

        public void addCardMin( String entity, String relation, String value ) {
                model.addCardMin( entity, relation, value );
        }

        public void addCardMax( String entity, String relation, String value ) {
                model.addCardMax( entity, relation, value );
        }

        public Object getModel() {
                return model;
        }
}
```

L'OrientedERBuilder è apparentemente più complesso perché deve produrre una stringa di testo con dati che sono ricevuti in qualunque ordine. Non è tanto importante il modo di costruire la stringa, come il fatto di osservare che questa classe fornisce codice soltanto per i metodi addRelationship, addCardMax e getModel, sufficienti per costruire il modello.

```
        import java.util.Enumeration;
        import java.util.Hashtable;

        public class OrientedERBuilder extends ModelBuilder {
                private Hashtable relations;
                public OrientedERBuilder() {
                relations = new Hashtable();
        }

        public void addRelationship(String fromEntity, String toEntity,
                        String relation ) {
                String[] relDetail = { fromEntity, toEntity, "0", "0" };
                relations.put( relation, relDetail );
        }

        public void addCardMax( String entity, String relation, String value ) {
                String[] relDetail=(String[]) relations.get( relation );
                if( entity.equals( relDetail[0] ) )
                        relDetail[3] = value;
```

```
                else
                        relDetail[2] = value;
                        relations.put( relation, relDetail);
        }

        public Object getModel() {
                String model ="";
                for(Enumeration elem = relations.elements();
                        elem.hasMoreElements() ; ) {
                        String[] currEl = (String[]) elem.nextElement() ;
                        model += "[ " + currEl[0] + " ]----("
                                        + currEl[2] + "," + currEl[3]
                                        + ")---->[ " + currEl[1]+ " ]\n";
                }
                return model;
        }
}
```

Il **Director** riceve un riferimento a un **ConcreteBuilder**, che dovrà eseguire le sue istruzioni per la costruire del modello. Il **Director** di questo esempio è implementato come una classe che ha il metodo statico GETMODEL, che fornisce la logica del processo di costruzione del modello. In una applicazione più realistica il Director potrebbe essere un oggetto capace di parificare un file o un flusso di dati, e a seconda i tokens riconosciuti, invocare i metodi del Builder. Si noti che il **Director** non è consapevole del **ConcreteBuilder** su cui agisce, dato che questo lo gestisce tramite l'interfaccia comune dei **Builder**, fornita dal **Builder** astratto (MODELBUILDER). Analogamente, il **Product** è trattato come un OBJECT.

```
public class ERHardCodedDirector {
        public static Object getModel( ModelBuilder builder ) {
                builder.addEntity( "Student" );
                builder.addEntity( "University" );
                builder.addEntity( "Professor" );
                builder.addRelationship( "Student", "University", "Studies at" );
                builder.addCardMin( "Student", "Studies at", "1" );
                builder.addCardMax( "Student", "Studies at", "1" );
                builder.addCardMin( "University", "Studies at", "0" );
                builder.addCardMax( "University", "Studies at", "N" );
                builder.addRelationship( "University", "Professor" , "Has" );
                builder.addCardMin( "University", "Has", "0" );
                builder.addCardMax( "University", "Has", "N" );
                builder.addCardMin( "Professor", "Has", "1" );
                builder.addCardMax( "Professor", "Has", "N" );
                return builder.getModel();
        }
}
```

La classe BUILDEREXAMPLE contiene il MAIN che fa la dimostrazione di questo *pattern*, realizzando una istanza di entrambi tipi di modelli. Si deve notare che il tipo di **Builder** concreto viene scelto a questo livello, e che in corrispondenza, a questo livello si fa il casting dei **Product** ritornati dal Director, verso le specifiche classi (nel primo caso STRING, e nel secondo, ERMODEL).

```
public class BuilderExample {
        public static void main( String[] arg ) {
                String swIngCourseModel = (String)
```

```
                ERHardCodedDirector.getModel( new OrientedERBuilder() );
                System.out.println( swIngCourseModel );
                ERModel dbCourseModel   = (ERModel)
                ERHardCodedDirector.getModel( new NotOrientedERBuilder() );
                dbCourseModel.showStructure();
        }
}
```

Finalmente si presenta il codice della classe di **Product** ERModel. Questa classe rappresenta il modello tramite collezioni d'oggetti Entity e Relationship, che sono d'uso interno della classe ERModel.

```
        import java.util.Enumeration;
        import java.util.Hashtable;
        public class ERModel {
                private Hashtable modelEntities = new Hashtable();
                private Hashtable modelRelations = new Hashtable();
                public void addEntity( String name ) {
                modelEntities.put( name, new Entity( name ) );
        }

        public void addRelationship(String entity1, String entity2,
                        String relation ) {
                Relationship rel = new Relationship();
                rel.name = relation;
                rel.entity1 = (Entity) modelEntities.get(entity1);
                rel.entity2 = (Entity) modelEntities.get(entity2);
                modelRelations.put( relation, rel );
        }

        public void addCardMin( String entity, String relation, String value ) {
                Relationship rel = (Relationship) modelRelations.get( relation );
                if( entity.equals( rel.entity1.name ) )
                        rel.cardMin1 = value;
                else
                        rel.cardMin2 = value;
        }

        public void addCardMax( String entity, String relation, String value ) {
                Relationship rel = (Relationship) modelRelations.get( relation );
                if( entity.equals( rel.entity1.name ) )
                        rel.cardMax1 = value;
                else
                        rel.cardMax2 = value;
                }

        public void showStructure( ) {
                for(Enumeration elem = modelRelations.elements();
                        elem.hasMoreElements() ; ) {
                Relationship currRel = (Relationship) elem.nextElement() ;
                System.out.println( "[ " + currRel.entity1.name +
                        " ]--("+ currRel.cardMin1 + "," +
                        currRel.cardMax1+
                        ")------< " + currRel.name +
                        " >------("+ currRel.cardMin2 + ","
```

```
C:\Design Patterns\Creational\Builder>java BuilderExample

(Il display dell'oggetto String costruito dal OrientedERBuilder)
[ Student ]----(N,1)---->[ University ]
[ University ]----(N,N)---->[ Professor ]

(L'esecuzione del metodo showModel dell'oggetto costruito dal
NotOrientedERBuilder)
[ Student ]--(1,1)------< Studies at >------(0,N)--[ University ]
[ University ]--(0,N)------< Has >------(1,N)--[ Professor ]
```

Figure 2.5: Esecuzione dell'esempio

```
                        + currRel.cardMax2 + ")--[ " +
                        currRel.entity2.name + " ]" );
                }
        }
}

class Entity {
        public String name;
        public Entity( String name ) {
                this.name = name;
        }
}

class Relationship {
        public String name;
        public Entity entity1, entity2;
        public String cardMin1, cardMax1, cardMin2, cardMax2 ;
}
```

Osservazioni sull'esempio

Per semplicità, il **Director** di questo esempio ha codificato dentro se stesso, in un metodo statico, il processo di creazione del modello. In una applicazione realistica il **Director** potrebbe essere un'oggetto (possibilmente un Singleton) in grado di parsificare di un file o di un'altra struttura dove risieda il modello di base.

2.5 Osservazioni sull'implementazione in Java

La classe astratta **Builder** (ModelBuilder) dichiara il metodo getModel che i **ConcreteBuilders** devono implementare, con il codice necessario per restituire ogni particolare tipo **Product**. Il tipo di ritorno del metodo getModel è indicato come Object, dato che a priori non si ha conoscenza della specifica tipologia di **Product**. In questo modo si abilita la possibilità di restituire qualunque tipo d'oggetto (perché tutte le classi Java, in modo diretto o indiretto, sono sottoclassi di Object). Si fa notare che il *"method overloading"* di Java non consente modificare la dichiarazione del tipo di valore di ritorno di un metodo di una sottoclasse, motivo per il quale i **ConcreteBuilders** devono anche dichiarare Object come valore di ritorno, nei propri metodi getModel.

Chapter 3

Prototype

3.1 Descrizione

Specifica i tipi di oggetti a creare, utilizzando un'istanza prototipo, e crea nuove istanze tramite la copia di questo prototipo. Java offre gli strumenti per strutturare una soluzione alternativa più flessibile al problema che diede origine a questo pattern, come verrà discusso più avanti.

3.2 Esempio

Si pensi alla creazione di uno schedulatore in grado di eseguire operazioni su oggetti che rappresentano istanti di tempo. Si ipotizzi che una delle operazioni da realizzare sia definire l'ora di termine di una attività, utilizzando come dati di input la sua ora di inizio e un numero intero corrispondente alla durata dell'attività. Questa operazione deve restituire un nuovo oggetto che rappresenti l'ora di termine, calcolata come la somma della ora di inizio e la durata dell'attività:

Si vuole progettare lo schedulatore in modo che sia in grado di ricevere come input (ora inizio) oggetti di diverse classi che implementino una particolare interfaccia. Si vuole che l'output dello schedulatore sia un oggetto contenente il risultato della schedulazione, che appartenga alla stessa classe dell'oggetto ricevuto come input. Il problema è che al momento di progettare lo schedulatore solo si ha conoscenza dell'interfaccia, non delle specifiche classi su cui dovrà agire.

3.3 Descrizione della soluzione offerta dal pattern

Gamma et Al. hanno proposto il "Prototype" pattern per situazioni di questo genere, che si basa sulla clonazione di oggetti utilizzati come prototipi. Questi oggetti devono implementare una interfaccia che offre un servizio di copia dell'oggetto, e che sarà quella di cui il framework avrà conoscenza al momento di dover creare nuovi oggetti. Il Prototype pattern è suggerito per essere applicato nel caso dei sistemi che devono essere indipendenti del modo in quale i loro prodotti sono creati, composti e rappresentati, e:

- quando le classi a istanziare sono specificate in run-time;
- quando si vuole evitare la costruzione di gerarchie di factories parallele alle gerarchie di prodotti;
- quando le istanze delle classi a istanziare hanno un insieme ridotto di stati possibili.

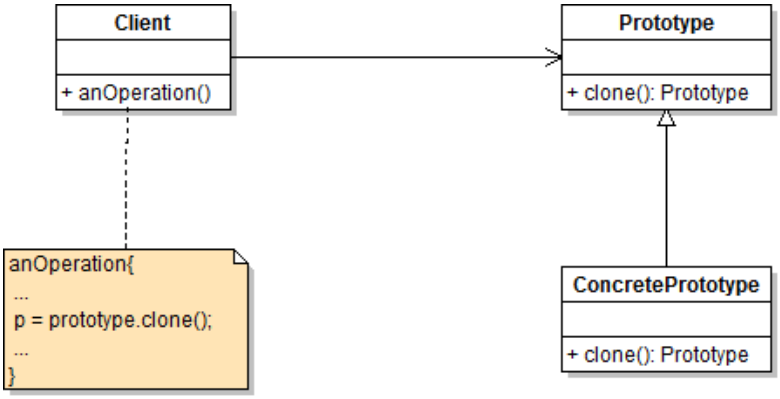


Figure 3.1: Struttura del Pattern

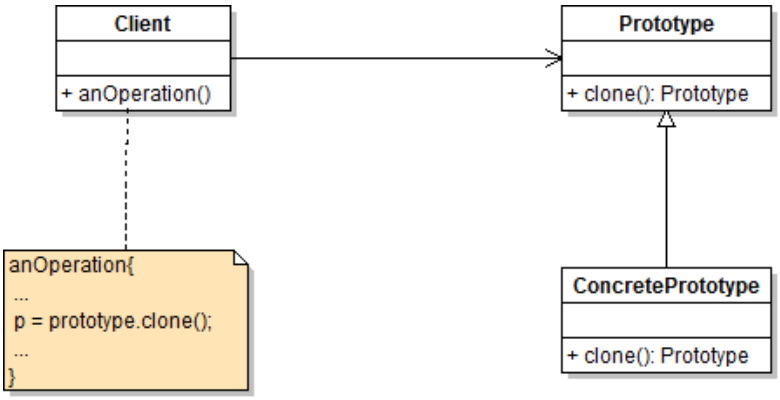


Figure 3.2: Schema del modello

Autori come Cooper[4] hanno adattato questo pattern a Java utilizzando il concetto di clonazione o copia tramite serializzazione, per risolvere il problema che ha dato origine al Pattern. Questo approccio sarà esemplificato di seguito, intanto che un secondo approccio, basato sulla Java Reflection API, sarà spiegato nella sezione Osservazioni sull'implementazione in Java.

3.4 Applicazione del modello

Partecipanti

- **Prototype**: classe astratta CloneableTime
  - Dichiara le implementa l'interfaccia per clonarsi da se (deve implementare l'interfaccia java.lang.cloneable per utilizzare il meccanismo di clonazione fornito da Java).
  - Implementa il metodo che verrà chiamato per clonare l'oggetto (cloneItself), il quale fa un richiamo al metodo protetto clone ereditato da java.lang.Object.
- **ConcretePrototype**: classi TimelmplementationC1 e TimelmplementationC2
  - Implementano le particolari versioni di oggetti da utilizzare e clonare (estendono Cloneable-Time).

- **Director**: classe SchedulerC.
  - Richiama il metodo di clonazione degli oggetti TimeImplementationC1 e TimeImplementationC2, per creare un nuovo oggetto.

Descrizione del codice

Il seguente codice presenta l'applicazione che fa uso dello schedulatore (SchedulerC) per fare un primo calcolo su di un oggetto della classe TimeImplementationC1, e un secondo calcolo su di un oggetto della classe TimeImplementationC2. Ognuno di questi oggetti mantiene al suo interno la rappresentazione dell'ora di inizio di una attività. Il risultato generato dallo schedulatore è un oggetto della stessa classe ricevuta come argomento.

```
public class PrototipeCloneExample {
        public static void main(String[] args) throws CloneNotSupportedException{
                System.out.println( "Using TimeImplementationC1:" );
                CloneableTime t1 = new TimeImplementationC1();
                t1.setTime( 15, 20, 10 );
                CloneableTime tEnd1 = SchedulerC.calculateEnd( t1 , 6 );
                System.out.println( "End: " + tEnd1.getHours() + ":" +
                        tEnd1.getMinutes() + ":" + tEnd1.getSeconds() );
                System.out.println( "Using TimeImplementationC2:" );
                CloneableTime t2 = new TimeImplementationC2();
                t2.setTime( 10, 15, 35 );
                CloneableTime tEnd2 = SchedulerC.calculateEnd( t2 , 6 );
                System.out.println( "End: " + tEnd2.getHours() + ":" +
                        tEnd2.getMinutes() + ":" + tEnd2.getSeconds() );
        }
}
```

Le classi alle quali appartengono gli oggetti che rappresentano le ore estendono la classe CloneableTime. Questa classe, oltre a definire l'interfaccia degli oggetti da utilizzare, implementa il metodo cloneItself che utilizza il meccanismo della clonazione di oggetti fornito da Java.

```
public abstract class CloneableTime implements Cloneable {
        public abstract void setTime(int hr, int min, int sec);

        public abstract int getHours();

        public abstract int getMinutes();

        public abstract int getSeconds();

        public CloneableTime cloneItself() throws CloneNotSupportedException {
                CloneableTime theClone = (CloneableTime) super.clone();
                theClone.setTime( 0, 0, 0 );
                return theClone;
        }
}
```

Le classi Java che utilizzano il meccanismo di clonazione devono implementare l'interfaccia java.lang.Cloneab e devono invocare il metodo privato clone ereditato dalla classe java.lang.Object. Le classi TimeImplementationC1 e TimeImplementationC2 estendono CloneableTime, si presentano a continuazione:

```
public class TimeImplementationC1 extends CloneableTime {
        private int hr, min, sec;

        public void setTime(int hr, int min, int sec) {
                this.hr = hr;
                this.min = min;
                this.sec = sec;
        }

        public int getHours() {
                return hr;
        }

        public int getMinutes() {
                return min;
        }

        public int getSeconds() {
                return sec;
        }
}
```

```
public class TimeImplementationC2 extends CloneableTime {
        private int secs;

        public void setTime(int hr, int min, int sec) {
                secs = hr * 3600 + min * 60 + sec;
        }

        public int getHours() {
                return secs / 3600;
        }

        public int getMinutes() {
                return (secs - getHours()*3600) / 60;
        }

        public int getSeconds() {
                return secs % 60;
        }
}
```

Lo schedulatore, implementato nella classe SchedulerC ha il metodo calcolateEnd che accetta come primo parametro una sottoclasse di CloneableTime, e come secondo parametro, un numero intero necessario per il calcolo dell'ora di termine. Questo metodo esegue il calcolo in base agli argomenti ricevuti, e restituisce come risposta una copia dell'oggetto ricevuto, tramite l'invocazione al suo metodo cloneItself, e salva in questo oggetto il risultato. Il riferimento a esso viene restituito alla fine.

```
public class SchedulerC {
        public static CloneableTime calculateEnd(CloneableTime start, int hours)
                        throws CloneNotSupportedException {
                int hr = start.getHours() + hours;
```

```
c:\Patterns\Creational\Prototype\Example1>java PrototypeCloneExample

Using TimeImplementationC1:
End: 21:20:10

Using TimeImplementationC2:
End: 16:15:35
```

Figure 3.3: Esecuzione dell'esempio

```
                hr = hr < 24 ? hr : hr - 24;
                CloneableTime endTime = start.cloneItself();
                endTime.setTime( hr, start.getMinutes(), start.getSeconds());
                return endTime;
        }
}
```

Osservazioni sull'esempio

In questo esempio si è voluto presentare l'utilizzo del Prototype pattern nel caso dell'istanziazione di classi in run-time.

3.5 Osservazioni sull'implementazione in Java

La Reflection API di Java offre strumenti più flessibili per la gestione della creazione di oggetti specificati in run-time, che quello rappresentato dal *Prototype* pattern. Nell'esempio che di seguito verrà presentato, la Reflection API viene utilizzata in un caso di schedulazione praticamente identico al problema presentato nella sezione precedente. Il diagramma di classi di questo esempio è il seguente:

Innanzitutto si presenta l'interfaccia TIME che dovranno implementare tutti gli oggetti sui quali dovrà lavorare lo schedulatore. Si noti che in questa versione l'interfaccia JAVA.LANG.CLONEABLE non viene utilizzata.

```
public interface Time {
        public void setTime(int hr, int min, int sec);

        public int getHours();

        public int getMinutes();

        public int getSeconds();
}
```

Si propongono le classi TIMEIMPLEMENTATIONR1 e TIMEIMPLEMENTATIONR2, analoghe alle TIMEIMPLEMENTATIONC1 e TIMEIMPLEMENTATIONC2 presentate in precedenza:

```
public class TimeImplementationR1 implements Time {
        private int hr, min, sec;

        public void setTime(int hr, int min, int sec) {
                this.hr = hr;
                this.min = min;
                this.sec = sec;
        }
```

```
        public int getHours() {
                return hr;
        }

        public int getMinutes() {
                return min;
        }

        ublic int getSeconds() {
                return sec;
        }
}
```

```
public class TimeImplementationR2 implements Time {
        private int secs;

        public void setTime(int hr, int min, int sec) {
                secs = hr * 3600 + min * 60 + sec;
        }

        public int getHours() {
                return secs / 3600;
        }

        public int getMinutes() {
                return (secs - getHours()*3600) / 60;
                // TODO: Add your code here
        }

        public int getSeconds() {
                return secs % 60;
        }
}
```

Si propone una terza implementazione di Time, la classe MaxTime, che verrà anche utilizzata nell'esempio (questa classe non fa altro che contenere il limite orario 23:59:59):

```
public class MaxTime implements Time {
        public void setTime(int hr, int min, int sec) {
                // Does nothing
        }

        public int getHours() {
                return 23;
        }

        public int getMinutes() {
                return 59;
        }

        public int getSeconds() {
                return 59;
        }
```

```
}
```

Lo schedulatore, rappresentato dalla classe SchedulerR ha la stessa funzionalità dello schedulatore descritto nell'esempio precedente (SchedulerC), l'unica differenza è che si serve della funzione TimeFactoryR.getNewTimeObject(Time) per ottenere una nuova istanza del tipo di oggetto Time ricevuto come parametro:

```
public class SchedulerR {
        public static Time calculateEnd( Time start, int hours )
                        throws TimeFactoryException {
                Time endTime = TimeFactoryR.getNewTimeObject( start );
                int hr = start.getHours() + hours;
                hr = hr < 24 ? hr : hr - 24;
                endTime.setTime( hr, start.getMinutes(), start.getSeconds());
                return endTime;
        }
}
```

La classe TimeFactory è una *utility* class che non fa altro che offrire la funzionalità di creare oggetti in base a due metodi che traggono vantaggio della Reflection API:

- public static Time getNewTimeObject(Time objectTime): che accetta come parametro un oggetto che li serve di riferimento per ottenere la Classe di oggetto da creare.
- public static Time getNewTimeObject (String timeClassName: che crea un oggetto della classe il cui nome (stringa) è ricevuto come parametro.

La classe TimeFactory si presenta di seguito:

```
import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;

public class TimeFactory {
        public static Time getNewTimeObject( Time objectTime )
                        throws TimeFactoryException {
                try {
                        Class timeClass = objectTime.getClass();
                        Constructor timeClassConstr =
                                timeClass.getConstructor( new Class[] { });
                        return (Time) timeClassConstr.newInstance(new Object[]{});
                } catch (NoSuchMethodException e) {
                        throw new TimeFactoryException( e );
                } catch (IllegalAccessException e) {
                        throw new TimeFactoryException( e );
                } catch (InvocationTargetException e) {
                        throw new TimeFactoryException( e );
                } catch (InstantiationException e) {
                        throw new TimeFactoryException( e );
                }
        }

        public static Time getNewTimeObject( String timeClassName )
                        throws TimeFactoryException {
                try {
                        Class timeClass = Class.forName( timeClassName );
```

```
                    Constructor timeClassConstr =
                            timeClass.getConstructor( new Class[] { });
                    return (Time) timeClassConstr.newInstance(new Object[]{});
                } catch (ClassNotFoundException e) {
                    throw new TimeFactoryException( e );
                } catch (NoSuchMethodException e) {
                    throw new TimeFactoryException( e );
                } catch (IllegalAccessException e) {
                    throw new TimeFactoryException( e );
                } catch (InvocationTargetException e) {
                    throw new TimeFactoryException( e );
                } catch (InstantiationException e) {
                    throw new TimeFactoryException( e );
                }
        }
}
```

Nel caso del primo dei metodi le istruzioni centrali sono le seguenti tre:

$$\begin{aligned} Class timeClass = objectTime.getClass(); \\ Constructor timeClassConstr = timeClass.getConstructor(newClass[]); \\ return (Time) timeClassConstr.newInstance(newObject[]); \end{aligned}$$

La procedura di creazione è la seguente[1]:

- Primo: Si ottiene un oggetto della classe JAVA.LANG.CLASS che rappesenta il tipo di oggetto ricevuto come parametro (tramite l'applicazione del metodo GETCLASS, ereditato da JAVA.LANG.OBJECT, applicato sull'oggetto time ricevuto come parametro). La classe JAVA.LANG.CLASS è una metaclasse che contiene le informazioni di ogni classe caricata nel programma.

- Secondo: si crea una istanza di JAVA.LANG.REFLECT.CONSTRUCTOR associato alla classe identificata nell'istruzione precedente, tramite l'invocazione al metodo GETCONSTRUCTOR dell'oggetto della classe JAVA.LANGCLASS. Questo metodo richiede come parametro un ARRAY di oggetti della classe JAVA.LANG.CLASS che rappresentano i tipi parametri richiesti dal costruttore. Nel caso degli oggetti TIME utilizzati i costruttori non hanno parametri, motivo per il quale l'ARRAY utilizzato è vuoto.

- Terzo: si fa una invocazione al metodo NEWINSTANCE sull'oggetto JAVA.LANG.REFLECT.CONSTRUCTOR creato nel passo precedente, tramite il quale si crea un nuovo oggetto della classe TIME. Questo metodo richiede come parametro un ARRAY di oggetti (dichiarato come JAVA.LANG.OBJECT) contenente i valori da fornire al costruttore. Nuovamente, dato che il costruttore non richiede parametri, l'ARRAY è istanziato vuoto. L'istanza, restituita da NEWISTANCE (un JAVA.LANG.OBJECT), è restituita dopo un casting a TIME dal metodo GETNEWTIMEOBJECT.

Finalmente si noti che tali istruzioni vengono racchiuse dentro un blocco TRY...CATCH che consente di gestire tute le eccezioni che si possono sollevare nella creazione degli oggetti. Per comodità si ha deciso di incapsulare la tipologia particolare di errore dentro di un unico tipo di oggetto errore chiamato TIMEFACTORYEXCEPTION, in modo di semplificare la gestione degli errori all'esterno della Factory.

[1] Si presentano senza maggior approfondimento le istruzioni che consentono di creare un oggetto in base ad un riferimento a un'altro oggetto, tramite la Java Reflection API. Una descrizione dettagliata si puo trovare in[15].

In modo analogo, il metodo della classe TimeFactory, che crea oggetti a partire del nome della classe, ha tre istruzione importanti, di cui la prima è solo diversa dalle descritte nei paragrafi precedenti:

$$\boxed{ClasstimeClass = Class.forName(timeClassName);}$$

In questo ultimo caso l'oggetto della classe java.lang.Class che rappresenta la classe da istanziare, viene creato dal metodo Class.forName(String). Il metodo forName restituisce un oggetto del tipo java.lang.Class che contiene le informazioni corrispondenti alla classe specifica il cui nome è indicato come parametro.

Si noti che le altre due istruzioni che seguono alla prima sono esattamente le stesse descritte precedentemente (punti identificati come "Secondo" e "Terzo").

La annunciata classe TimeFactoryException è implementata in questo modo:

```
public class TimeFactoryException extends Exception {
        public TimeFactoryException( Exception e ) {
                super( e.getMessage() );
        }
}
```

Finalmente si presenta il codice dell'esempio che fa una prima dimostrazione dell'uso delle classi TimeImplementationR1 e TimeImplementationR2 insieme allo schedulatore, e una dimostrazione della creazione di un oggetto, tramite l'indicazione del nome della classe.

```
public class ReflectiveCloneExample {
        public static void main(String[] args) throws TimeFactoryException {
                //***** Works with the TimeImplementationR1 class
                TimeImplementationR1 t1 = new TimeImplementationR1();
                t1.setTime( 15, 20, 10 );
                Time tEnd1 = SchedulerR.calculateEnd( t1 , 6 );
                System.out.print( "End: " + tEnd1.getHours() + ":"
                        + tEnd1.getMinutes() +
                        ":" + tEnd1.getSeconds() );
                //Prints the class name
                System.out.println( " ...using " + tEnd1.getClass() );
                //***** Works with the TimeImplementationR2 class
                TimeImplementationR2 t2 = new TimeImplementationR2();
                t2.setTime( 10, 15, 35 );
                Time tEnd2 = SchedulerR.calculateEnd( t2 , 6 );
                System.out.print( "End: " + tEnd2.getHours() + ":"
                        + tEnd2.getMinutes() +
                        ":" + tEnd2.getSeconds() );
                //Prints the class name
                System.out.println( " ...using " + tEnd2.getClass() );
                //***** Loads a class specified by name (run-time)
                Time lastTime = TimeFactoryR.getNewTimeObject( "MaxTime" );
                System.out.print( "Max: " + lastTime.getHours() + ":"
                        + lastTime.getMinutes() +
                        ":" + lastTime.getSeconds() );
                //Prints the class name
                System.out.println( " ...using " + lastTime.getClass() );
        }
}
```

```
c:\Patterns\Creational\Prototype\Example2>java ReflectiveCloneExample

End: 21:20:10 ...using class TimeImplementationR1
End: 16:15:35 ...using class TimeImplementationR2
Max: 23:59:59 ...using class MaxTime
```

Figure 3.4: Esecuzione dell'esempio

Nota

Si osservi che la costruzione degli oggetti tramite la Reflection API deve gestire gli stesi costruttori che le classi da istanziare possiedono (coincidenza di parametri), ma questo non può essere a priori assicurato.

Chapter 4

Adapter

4.1 Descrizione

Converte l'interfaccia di una classe in un'altra interfaccia aspettata dai clienti. In questo modo, si consente la collaborazione tra classi che in un altro modo non potrebbero interagire dovuto alle loro diverse interfacce.

4.2 Esempio

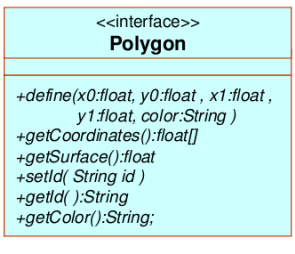
Si vuole sviluppare un'applicazione per lavorare con oggetti geometrici. Questi oggetti saranno gestiti dall'applicazione tramite un'interfaccia particolare (Polygon), che offre un insieme di metodi che gli oggetti grafici devono implementare. A questo punto si ha a disposizione una antica classe (Rectangle) che si potrebbe riutilizzare, che però ha un'interfaccia diversa, e che non si vuole modificare.

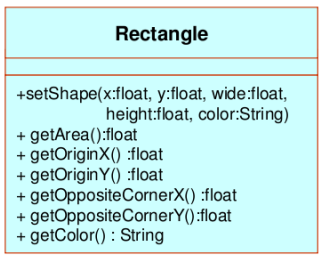
Il problema consiste nella definizione di un modo di riutilizzare la classe esistente tramite una nuova interfaccia, ma senza modificare l'implementazione originale.

4.3 Descrizione della soluzione offerta dal pattern

L'Adapter pattern offre due soluzioni possibili, denominate Class Adapter e Object Adapter, che si spiegano di seguito:

- Class Adapter: la classe esistente si estende in una sottoclasse (RectangleClassAdapter) che implementa la desiderata interfaccia. I metodi della sottoclasse mappano le loro operazioni in richieste ai metodi e attributi della classe di base.





New interface Available class

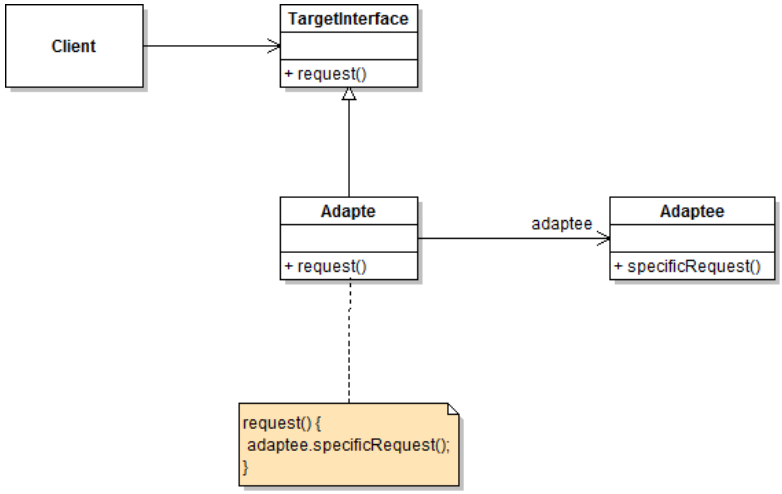


Figure 4.1: Struttura del Pattern per il Class Adapter

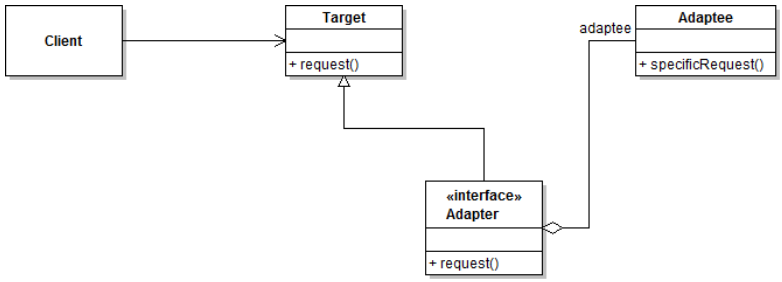


Figure 4.2: Struttura del Pattern per l'Object Adapter

- Object Adapter: si crea una nuova classe (RectangleObjectAdapter) che implementa l'interfaccia richiesta, e che possiede al suo interno un'istanza della classe a riutilizzare. Le operazioni della nuova classe fanno invocazioni ai metodi dell'oggetto interno.

4.4 Struttura del pattern

Partecipanti

- **TargetInterface**: interfaccia Polygon.
  - Specifica l'interfaccia che il **Client** utilizza.
- **Client**: classi ClassAdapterExample e ObjectAdapterExample.
  - Comunica con l'oggetto interessato tramite la **TargetInterface**.
- **Adaptee**: interfacce Rectangle.
  - Implementa una interfaccia che deve essere adattata.
- **Adapter**: classi RectangleClassAdapter, RectangleObjectAdapter.
  - Adatta l'interfaccia dell'**Adaptee** verso la **TargetInterface**.

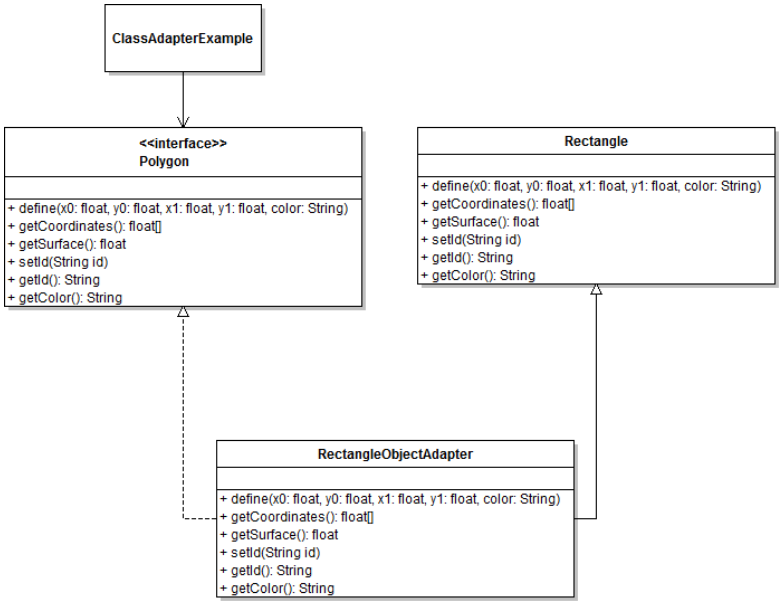


Figure 4.3: Schema del modello per il Class Adapter

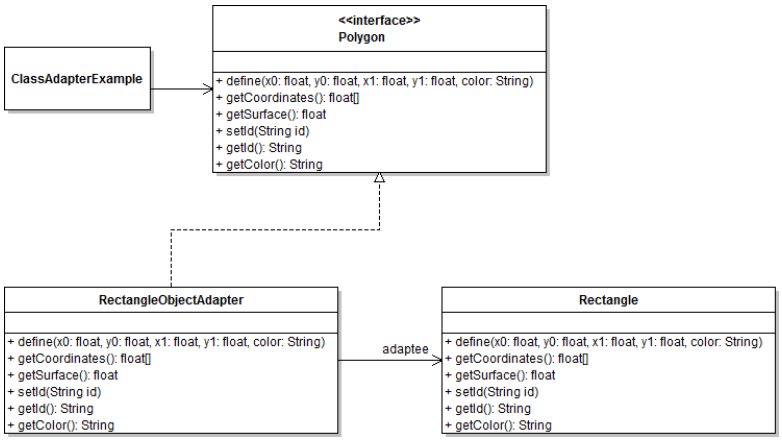


Figure 4.4: Schema del modello per l'Object Adapter

Implementazione

Si presenteranno esempi di entrambi tipi d'Adapter. In ogni caso il problema sarà la riutilizzazione della classe Rectangle in un'applicazione che crei oggetti che implementano l'interfaccia POLYGON.

La classe RECTANGLE è implementata in questo modo:

```
public class Rectangle {
        private float x0, y0;
        private float height, width;
        private String color;

        public void setShape(float x, float y, float a, float l, String c) {
                x0 = x;
                y0 = y;
                height = a;
                width = l;
                color = c;
        }

        public float getArea() {
                return x0 * y0;
        }

        public float getOriginX() {
                return x0;
        }

        public float getOriginY() {
                return y0;
        }

        public float getOppositeCornerX() {
                return x0 + height;
        }

        public float getOppositeCornerY() {
                return y0 + width;
        }

        public String getColor() {
                return color;
        }
}
```

Invece l'interfaccia POLYGON si deve utilizzare è questa:

```
public interface Polygon {
        public void define( float x0, float y0, float x1, float y1,
                String color );
        public float[] getCoordinates() ;
        public float getSurface();
        public void setId( String id );
        public String getId( );
        public String getColor();
}
```

Si osservi che le caratteristiche del rettangolo (classe RECTANGLE) vengono indicate nel metodo SETSHAPE, che riceve le coordinate del vertice superiore sinistro, l'altezza, la larghezza e il colore. Dall'altra parte, l'interfaccia POLYGON specifica che la definizione delle caratteristiche della figura, avviene tramite il metodo chiamato DEFINE, che riceve le coordinate degli angoli opposti e il colore.

Si osservi che il metodo GETCOORDINATES dell'interfaccia POLYGON restituisce un array contenente le coordinate degli angoli opposti (nel formato x0, y0, x1, y1), intanto nella classe RECTANGLE esistono metodi particolari per ricavare ogni singolo valore (GETORIGINX, GETORIGINY, GETOPPOSITECORNERX e GETOPPOSITECORNERY).

In quel che riguarda la superficie del rettangolo, in entrambi casi si ha a disposizione un metodo, ma con nome diverso.

Si noti, anche, che POLYGON aggiunge metodi non mappabili sulla classe RECTANGLE (SETID e GETID), che consentono la gestione di un identificativo tipo STRING per ogni figura creata.

Finalmente si noti che il metodo GETCOLOR ha la stessa firma e funzione nell'interfaccia POLYGON e nella classe RECTANGLE.

Di seguito si descrivono le due implementazioni proposte.

- Implementazione come Class Adapter La costruzione del Class Adapter per il RECTANGLE è basato nella sua estensione. Per questo obbiettivo viene creata la classe RECTANGLECLASSADAPTER che estende RECTANGLE e implementa l'interfaccia POLYGON:

```
public class RectangleClassAdapter extends Rectangle implements Polygon{
        private String name = "NO NAME";

        public void define( float x0, float y0, float x1, float y1,
                        String color ) {
                float a = x1 - x0;
                float l = y1 - y0;
                setShape( x0, y0, a, l, color );
        }

        public float getSurface() {
                return getArea();
        }

        public float[] getCoordinates() {
                float aux[] = new float[4];
                aux[0] = getOriginX();
                aux[1] = getOriginY();
                aux[2] = getOppositeCornerX();
                aux[3] = getOppositeCornerY();
                return aux;
        }

        public void setId( String id ) {
                name = id;
        }

        public String getId( ) {
                return name;
        }
}
```

Si noti che la funzionalità riguardante l'identificazione del rettangolo, non presenti nella classe di base, si implementa completamente nella classe **Adapter**.

Le altre funzionalità si ottengono fornendo le operazioni particolari necessarie che richiamando i metodi della classe RECTANGLE.

Si noti che il metodo GETCOLOR è ereditato dalla classe di base.

Il **Client** di questo rettangolo adattato è la classe CLASSADAPTEREXAMPLE:

```
public class ClassAdapterExample {
        public static void main( String[] arg ) {
                Polygon block = new RectangleClassAdapter();
                block.setId( "Demo" );
                block.define( 3 , 4 , 10, 20, "RED" );
                System.out.println( "The area of "+ block.getId() + " is "+
                block.getSurface() + ", and its " +
                block.getColor() );
        }
}
```

- Implementazione come Object Adapter La costruzione dell'Object Adapter per il RECTANGLE, si basa nella creazione di una nuova classe (RectangleObjectAdapter) che avrà al suo interno un'oggeto della classe RECTANGLE, e che implementa l'interfaccia POLYGON:

```
public class RectangleObjectAdapter implements Polygon {
        Rectangle adaptee;

        private String name = "NO NAME";
                public RectangleObjectAdapter() {
                adaptee = new Rectangle();
        }

        public void define( float x0, float y0, float x1, float y1,
                        String col ) {
                float a = x1 - x0;
                float l = y1 - y0;
                adaptee.setShape( x0, y0, a, l, col);
        }

        public float getSurface() {
                return adaptee.getArea();
        }

        public float[] getCoordinates() {
                float aux[] = new float[4];
                aux[0] = adaptee.getOriginX();
                aux[1] = adaptee.getOriginY();
                aux[2] = adaptee.getOppos iteCornerX();
                aux[3] = adaptee.getOppositeCornerY();
                return aux;
        }

        public void setId( String id ) {
                name = id;
        }
```

```
        public String getId( ) {
                return name;
        }

        public String getColor( ) {
                return adaptee.getColor();
        }
}
```

Si noti come in questo caso la costruzione di un RectangleObjectAdapter porta con se la creazione al suo interno di un oggetto della classe Rectangle. Ecco il codice del **Client** (ObjectAdapterExample) che fa uso di questo **Adapter**:

```
public class ObjectAdapterExample {
        public static void main( String[] arg ) {
                Polygon block = new RectangleObjectAdapter();
                block.setId( "Demo" );
                block.define( 3 , 4 , 10, 20, "RED" );
                System.out.println( "The area of "+ block.getId() + " is "+
                block.getSurface() + ", and it's " +
                block.getColor() );
        }
}
```

Osservazioni sull'esempio

In questo esempio si ha dimostrato l'Adapter pattern, considerando un caso nel quale l'interfaccia richiesta:

- Ha un metodo la cui funzionalità si può ottenere dall'**Adaptee**, previa esecuzione di alcune operazioni (metodo define).
- Ha un metodo la cui funzionalità si può ottenere dall'**Adaptee** tramite l'invocazione di un'insieme dei suoi metodi (getCoordinates).
- Ha un metodo la cui funzionalità si ricava direttamente da un metodo dell'**Adaptee**, che ha soltanto una firma diversa (metodo getSurface).
- Ha un metodo la cui funzionalità si ricava direttamente da un metodo dell'**Adaptee**, e con la stessa firma (getColor).
- Ha metodi che aggiungono nuove operazioni che non si ricavano dai metodi dell'**Adaptee** (setId e getId).

4.5 Osservazioni sull'implementazione in Java

La strategia di costruire un **Class Adapter** è possibile soltanto se l'**Adaptee** non è stato dichiarato come final class.

```
C:\Design Patterns\Structural\Adapter>java ClassAdapterExample
The area of Demo is 12.0, and it's RED

C:\Design Patterns\Structural\Adapter>java ObjectAdapterExample
The area of Demo is 12.0, and it's RED
```

Figure 4.5: Esecuzione dell'esempio

Chapter 5

Composite

5.1 Descrizione

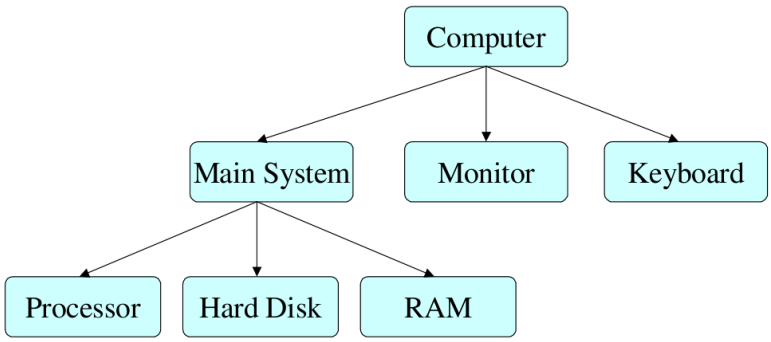
Consente la costruzione di gerarchie di oggetti composti. Gli oggetti composti possono essere conformati da oggetti singoli, oppure da altri oggetti composti. Questo pattern è utile nei casi in cui si vuole:

- Rappresentare gerarchie di oggetti tutto-parte.
- Essere in grado di ignorare le differenze tra oggetti singoli e oggetti composti.

5.2 Esempio

Nel magazzino di una ditta fornitrice di computer ci sono diversi prodotti, quali computer pronti per la consegna, e pezzi di ricambio (o pezzi destinati alla costruzione di nuovi computer). Dal punto di vista della gestione del magazzino, alcuni di questi pezzi sono pezzi singoli (indivisibili), altri sono pezzi composti da altri pezzi. Ad esempio, il "monitor", la "tastiera" e la "RAM" sono pezzi singoli, intanto il "main system", è un pezzo composto da tre pezzi singoli ("processore", "disco rigido" e "RAM"). Un altro esempio di pezzo composto è il "computer", che si compone di un pezzo composto ("main system"), e due pezzi singoli ("monitor" e "tastiera").

Il problema è la rappresentazione omogenea di tutti gli elementi presenti del magazzino, sia dei singoli componenti, sia di quelli composti da altri componenti.



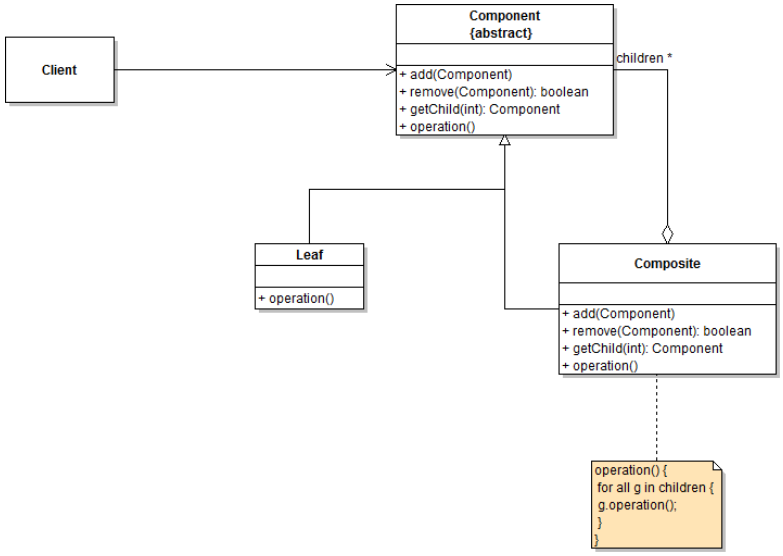


Figure 5.1: Struttura del Pattern

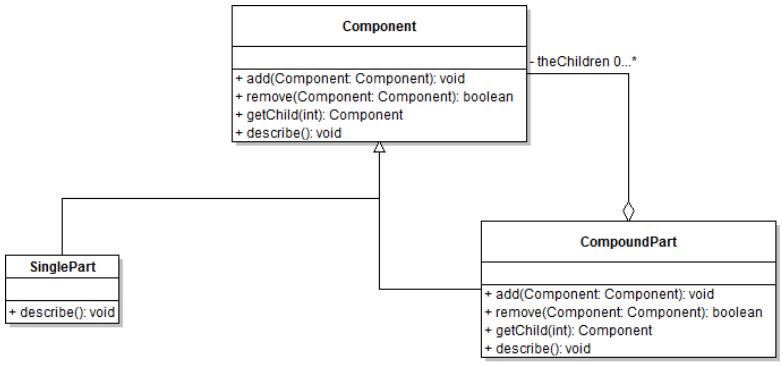


Figure 5.2: Schema del modello

5.3 Descrizione della soluzione offerta dal pattern

Il pattern "Composite" definisce la classe astratta componente (Component) che deve essere estesa in due sottoclassi: una che rappresenta i singoli componenti (Leaf), e un'altra (Composite) che rappresenta i componenti composti, e che si implementa come contenitore di componenti. Il fatto che quest'ultima sia un contenitore di componenti, li consente di immagazzinare al suo interno, sia componenti singoli, sia altri contenitori (dato che entrambi sono stati dichiarati come sottoclassi di componenti).

5.4 Applicazione del Pattern

Partecipanti

- **Component**: classe astratta COMPONENT.

  - Dichiara una interfaccia comune per oggetti singoli e composti.
  - Implementa le operazioni di default o comuni tutte le classi.

- **Leaf**: classe SinglePart.
  - Estende la classe Component, per rapperesentare gli oggetti che non sono composti (foglie).
  - Implementa le operazioni per questi oggetti.

- **Composite**: classe CompoundPart.
  - Estende la classe Component, per rappresentare gli oggetti che sono composti.
  - Immagazzina al suo interno i propri componenti.
  - Implementa le operazioni proprie degli oggetti composti, e particolarmente quelle che riguardano la gestione dei propri componenti.

- **Client**: in questo esempio sarà il programma principale quello che farà le veci di cliente.
  - Utilizza gli oggetti singoli e composti tramite l'interfaccia rappresentata dalla classe astratta Component.

Descrizione del codice

La classe astratta Component definisce l'interfaccia comune di oggetti singoli e composti, e implementa le loro operazioni di default. Particolarmente le operazioni add(Component c) e remove(Component c) sollevano una eccezione del tipo SinglePartException se vengono invocate su un oggetto foglia (tentativo di aggiungere o rimuovere un componente). Invece nel caso di getChild(int n), che serve a restituire il componente di indice n, l'operazione di default restituisce null (questa è stata una scelta di progettazione, un'altra possibilità era sollevare anche in questo caso una eccezione)[1]. Il metodo describe() è dichiarato come metodo astratto, da implementare in modo particolare nelle sottoclassi. Il Costruttore di Component riceve una stringa contenete il nome del componente, che verrà assegnato ad ognuno di essi.

```
public abstract class Component {
        public String name;

        public Component(String aName){
                name = aName;
        }

        public abstract void describe();

        public void add(Component c) throws SinglePartException {
                if (this instanceof SinglePart)
                        throw new SinglePartException( );
        }

        public void remove(Component c) throws SinglePartException{
                if (this instanceof SinglePart)
                        throw new SinglePartException( );
```

[1] Questo modo di trattare eventuali tentativi di l'invocazione di metodi legati a oggetti composti, sulle foglie, è anche applicato da Landini[10].

```
        }

        public Component getChild(int n){
                return null;
        }
}
```

La classe SinglePart estende la classe Component. Possiede un costruttore che consente l'assegnazione del nome del singolo pezzo, il quale che verrà immagazzinato tramite l'invocazione al costruttore della superclasse. La classe SinglerPart fornisce, anche, l'implementazione del metodo describe().

```
public class SinglePart extends Component {
        public SinglePart(String aName) {
                super(aName);
        }

        public void describe(){
                System.out.println( "Component: " + name );
        }
}
```

La classe CompoundPart estende anche Component, e implementa sia i metodi di gestione dei componenti (add, remove, getChild), sia il metodo describe(). Si noti che il metodo describe() stampa in primo luogo il proprio nome dell'oggetto, e poi scandisce l'elenco dei suoi componenti, invocando il metodo describe() di ognuno di essi. Il risultato sarà che insieme alla stampa del proprio nome dell'oggetto composto, verranno anche stampati i nomi dei componenti.

```
import java.util.Vector;
import java.util.Enumeration;

public class CompoundPart extends Component {
        private Vector children ;

        public CompoundPart(String aName) {
                super(aName);
                children = new Vector();
        }

        public void describe(){
                System.out.println("Component: " + name);
                System.out.println("Composed by:");
                System.out.println("{");
                int vLength = children.size();
                for( int i=0; i< vLength ; i ++ ) {
                        Component c = (Component) children.get( i );
                        c.describe();
                }
                System.out.println("}");
        }

        public void add(Component c) throws SinglePartException {
                children.addElement(c);
        }
```

```
        public void remove(Component c) throws SinglePartException{
                children.removeElement(c);
        }

        public Component getChild(int n) {
                return (Component)children.elementAt(n);
        }
}
```

Si noti che in questa implementazione ogni CompoundPart gestisce i propri componenti in un Vector. La classe SinglePartException rappresenta l'eccezione che verrà sollevata nel caso che le operazioni di gestione dei componenti vengano invocate su una parte singola.

```
class SinglePartException extends Exception {
        public SinglePartException( ){
                super( "Not supported method" );
        }
}
```

L'applicazione CompositeExample fa le veci del Client che gestisce i diversi tipi di pezzi, tramite l'interfaccia comune fornita dalla classe Component. Nella prima parte dell'esecuzione si creano dei pezzi singoli (monitor, keyboard, processor, ram e hardDisk), dopodiché viene creato un oggetto composto (mainSystem) con tre di questi oggetti singoli. L'oggetto composto appena creato serve, a sua volta, per creare, insieme ad altri pezzi singoli, un nuovo oggetto composto (computer). L'applicazione invoca poi il metodo describe() su un oggetto singolo, sull'oggetto composto soltanto da pezzi singoli, e sull'oggetto composto da pezzi singoli e pezzi composti. Finalmente fa un tentativo di aggiungere un componente ad un oggetto corrispondente a un pezzo singolo.

```
public class CompositeExample {
        public static void main(String[] args) {
                // Creates single parts
                Component monitor       = new SinglePart("LCD Monitor");
                Component keyboard      = new SinglePart("Italian Keyboard");
                Component processor     = new SinglePart("Pentium III Processor");
                Component ram   = new SinglePart("256 KB RAM");
                Component hardDisk      = new SinglePart("40 Gb Hard Disk");

                // A composite with 3 leaves
                Component mainSystem = new CompoundPart( "Main System" );
                try {
                        mainSystem.add( processor );
                        mainSystem.add( ram );
                        mainSystem.add( hardDisk );
                }
                catch (SinglePartException e){
                        e.printStackTrace();
                }

                // A Composite compound by another Composite and one Leaf
                Component computer = new CompoundPart("Computer");
                        try{
                                computer.add( monitor );
                                computer.add( keyboard );
```

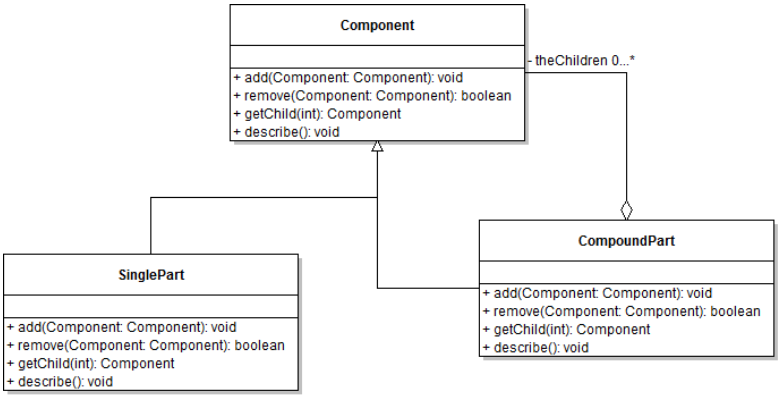


Figure 5.3:

```
                        computer.add( mainSystem );
                }
                catch (SinglePartException e){
                        e.printStackTrace();
                }

                System.out.println("**Tries to describe the monitor component");
                monitor.describe();
                System.out.println("Tries to describe the main system component");
                mainSystem.describe();
                System.out.println("**Tries to describe" +
                        + "the computer component" );
                computer.describe();

                // Wrong: invocation of add() on a Leaf
                System.out.println( "**Tries to add a component" +
                        + "to a single part(leaf) " );
                try{
                        monitor.add( mainSystem );
                }
                        catch (SinglePartException e){
                        e.printStackTrace();
                }
        }
}
```

Osservazioni sull'esempio

Si noti che nell'esempio presentato, la classe astratta COMPONENT fornisce un'implementazione di default per i metodi di gestione dei componenti (ADD, REMOVE, GETCHILD). Dal punto di vista del Composite pattern, sarebbe anche valida la dichiarazione di questi metodi come metodi astratti, lasciando l'implementazione alle classi SINGLEPART e COMPOUNDPART, come si può apprezzare nella figura 5.3.

Se si implementa il pattern in questo modo, si devono modificare le classi COMPONENT e SINGLEPART. In particolare, il codice della classe Component dovrebbe dichiarare i metodi di gestione

dei componenti (ADD, REMOVE e GETCHILD), come metodi astratti:

```
public abstract class Component {
        public String name;

        public Component(String aName){
        name = aName;
        }

        public abstract void describe();

        public abstract void add(Component c) throws SinglePartException;

        public abstract void remove(Component c) throws SinglePartException;

        public abstract Component getChild(int n);
}
```

E la classe SINGLEPART dovrebbe implementare il codice riguardante tutti i metodi dichiarati astratti nella superclasse:

```
public class SinglePart extends Component {
        public SinglePart(String aName) {
                super(aName);
        }

        public void add(Component c) throws SinglePartException{
                throw new SinglePartException( );
        }

        public void remove(Component c) throws SinglePartException{
                throw new SinglePartException( );
        }

        public Component getChild(int n) {
                return null;
        }

        public void describe(){
                System.out.println( "Component: " + name );
        }
}
```

5.5 Osservazioni sull'implementazione in Java

Non ci sono aspetti particolari da tenere in conto.

```
C:\Design Patterns\Structural\Composite >java CompositeExample

** Tries to describe the 'monitor' component
Component: LCD Monitor

** Tries to describe the 'main system' component
Component: Main System
Composed by:
{
Component: Pentium III Processor
Component: 256 KB RAM
Component: 40 Gb Hard Disk
}

** Tries to describe the 'computer' component
Component: Computer
Composed by:
{
Component: LCD Monitor
Component: Italian Keyboard
Component: Main System
Composed by:
{
Component: Pentium III Processor
Component: 256 KB RAM
Component: 40 Gb Hard Disk
}
}

** Tries to add a component to a single part (leaf)
SinglePartException: Not supported method
        at SinglePart.add(SinglePart.java:9)
        at CompositeExample.main(CompositeExample.java:46)
```

Figure 5.4: Esecuzione dell'esempio

Chapter 6

Proxy

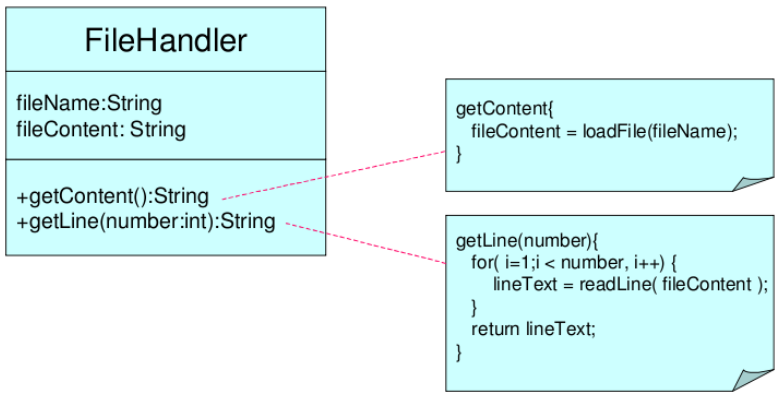
6.1 Descrizione

Fornisce una rappresentazione di un oggetto di accesso difficile o che richiede un tempo importante per l'accesso o creazione. Il Proxy consente di posticipare l'accesso o creazione al momento in cui sia davvero richiesto

6.2 Esempio

Un programma di visualizzazione di testi deve gestire informazioni riguardanti file dove i testi vengono archiviati, come il contenuto stesso dei file. Il programma potrebbe essere in grado di visualizzare il nome di un file, il testo completo, o trovare e visualizzare una singola riga. Si pensi che per l'implementazione si ha progettato un oggetto che al momento della sua istanziazione fa il caricamento del file, e che svolge le operazioni descritte nel seguente modo:

- Restituire nome del file: restituisce una stringa contenente il nome del file dentro il file system.
- Restituire il testo completo: restituisce una stringa contenente il testo.
- Restituire una singola riga di testo: riceve come parametro un numero valido di riga (si considera CR + LF come separatore di riga), e tramite un algoritmo di ricerca, restituisce una stringa contenente il testo corrispondente.
-



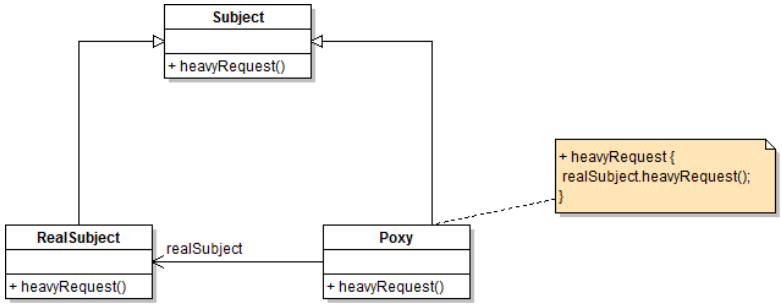


Figure 6.1: Struttura del Pattern

Questa classe diventa utile dato che fornisce tutte le funzionalità richieste dall'applicativo. Nonostante ciò, il suo uso è inefficiente perché se solo si vuole accedere al nome del file non è necessario aver caricato tutto il suo contenuto in memoria. Un altro caso di inefficienza si presenta nel caso in cui si fanno due richieste successive dello stesso numero di riga, che determinano la ripetizione della ricerca appena effettuata. E' di interesse poter gestire più efficientemente questa classe, senza modificare la sua implementazione.

6.3 Descrizione della soluzione offerta dal pattern

Il "Proxy" pattern suggerisce l'implemetazione di una classe (ProxyFileHandler) che offra la stesa interfaccia della classe originale (FileHandler), e che sia in grado di risolvere le richieste più "semplici" pervenute dall'applicativo, senza dover utilizzare inutilmente le risorse (ad esempio, restituire il nome del file). Solo al momento di ricevere una richiesta più "complessa" (ad esempio, restituire il testo del file), il proxy andrebbe a creare il vero FileHandler per inoltrare a esso le richieste. In questo modo gli oggetti più pesanti sono creati solo al momento di essere necessari. Il proxy che serve a questa finalità spesso viene chiamato "virtual proxy". Come altra funzionalità, a questo proxy potrebbe essere aggiunta la possibilità di imagazinare temporaneamente l'ultima riga restituita dal metodo "getLine", in modo che due richieste successive della stesa linea comportino solo una ricerca, riducendo lo spreco di tempo di elaborazione. Il proxy che offre questa funzionalità viene chiamato "caché proxy".

6.4 Applicazione del Pattern

Partecipanti

- **Proxy**: classe ProxyFileHandler.
  - Mantiene un riferimento per accedere al **RealSubject**.
  - Implementa una interfaccia identica a quella del **RealSubject**, in modo che può sostituire a esso.
  - Controlla l'acceso al RealSubject, essendo responsabile della sua istanziazione e gestione di riferimenti.
  - Come *virtual proxy* pospone la istanziazione del **RealSubject**, tramite la gestione di alcune informazioni di questo.
  - Come *caché proxy* immagazzina temporaneamente il risultato di alcune elaborazioni del **RealSubject**, in modo di avere delle risposte pronte per i clienti.

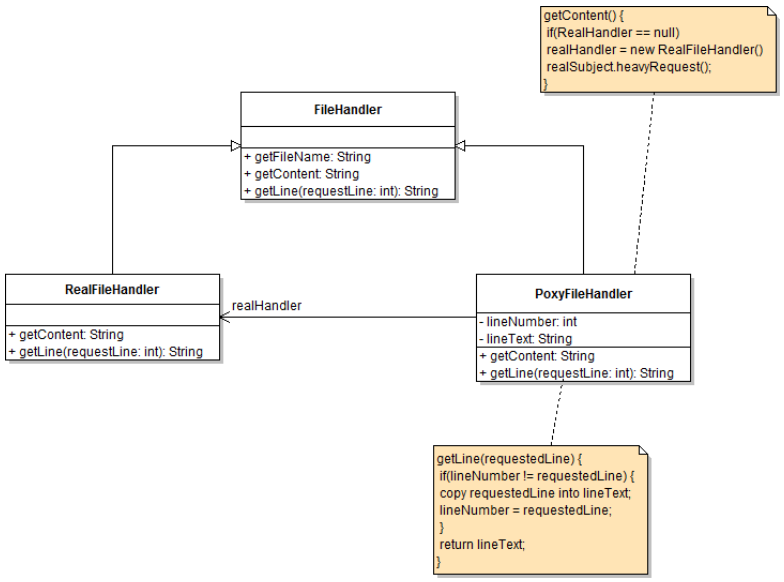


Figure 6.2: Schema del modello

- **Subject**: classe FileHandler.
  - Fornisce l'interfaccia comune per il **RealSubject** e il **Proxy**, in modo che questo ultimo possa essere utilizzato in ogni luogo dove si aspetta un **RealSubject**.
- **RealSubject**: classe RealFileHandler.
  - Implementa l'oggetto vero e proprio che il **RealSubject** rappresenta.

Descrizione del codice

Si crea la classe FileHandler che rappresenta l'interfaccia che la classe di gestione dei file (RealFileHandler) e il suo proxy (ProxyFileHandler) devono implementare. Questa classe offre la funzionalità di gestione del nome del file da aprire.

```
public abstract class FileHandler {
        protected String fileName;

        public FileHandler(String fName) {
                fileName = fName;
        }

        public String getFileName() {
                return fileName;
        }

        public abstract String getContent();

        public abstract String getLine( int requestedLine );
}
```

La classe RealFileHandler estende FileHandler. Nel costruttore viene fornito il codice di caricamento del file di testo in memoria. Questa classe offre i metodi di restituzione del testo del

file come una singola stringa, e di ricerca e restituzione di una singola riga di testo, a partire del suo numero.

```
import java.io.*;

class RealFileHandler extends FileHandler {
        private byte[] content;

        public RealFileHandler( String fName ){
                super(fName);
                System.out.println("(creating a real handler with file content)");
        FileInputStream file;
        try{
                file = new FileInputStream(fName);
                int numBytes = file.available();
                content = new byte[ numBytes ];
                file.read( content );
        } catch(Exception e){
        System.out.println( e );
        }
}

        public String getContent( ){
                return new String( content );
        }

        public String getLine( int requestedLine ){
                System.out.println( "(accessing from real handler)" );
                int numBytes = content.length;
                int currentLine = 1;
                int startingPos = -1;
                int lineLength = 0;
                for(int i=0;i<numBytes; i++) {
                        if((currentLine == requestedLine) && (content[i] != 0x0A)){
                                if( startingPos == -1)
                                        startingPos = i;
                                lineLength++;
                        }
                if( content[i] == 0x0D )
                        currentLine++;
                }
                String lineText = "";
                if(startingPos != -1)
                        lineText = new String(content, startingPos, lineLength-1);
                return "\"" + lineText + "\"";
        }
}
```

Si noti che nell'implementazione della classe RealFileHandler il testo del file viene caricato in memoria appena viene istanziata, e che l'algoritmo di ricerca di una riga viene eseguito ad ogni chiamata del metodo di restituzione di righe. La classe ProxyFileHandler implementa il proxy per il RealFileHandler. Questa classe, eredita la sua interfaccia e la gestione del nome del file associato, dalla superclasse FileHandler. Si noti che un oggetto della classe ProxyFileHandler crea un una istanza di RealFileHandler solo al momento in cui diventa indispensabile caricare in

memoria il contenuto del file, cioè la prima volta che viene richiesto il suo contenuto completo, o quello di una singola riga.

```
public class ProxyFileHandler extends FileHandler {
        private RealFileHandler realHandler;
        private int lineNumber;
        private String lineText;

        public ProxyFileHandler ( String fName ){
                super( fName );
                System.out.println( "(creating a proxy cache)" );
        }

        public String getContent(){
                if( realHandler == null )
                        realHandler = new RealFileHandler( fileName );
                return realHandler.getContent();
        }

        public String getLine(int requestedLine){
                if( requestedLine == lineNumber ) {
                        System.out.println( "(accessing from proxy cache)" );
                        return lineText;
                } else {
                if( realHandler == null )
                        realHandler = new RealFileHandler( fileName );
                        lineText = realHandler.getLine( requestedLine );
                        lineNumber = requestedLine;
                }
                return lineText;
        }
}
```

Si noti che in entrambi classi sono stati inseriti dei messaggi di testo che servono a monitorare le chiamate di ogni metodo, del RealFileHandler e del ProxyFileHandler. Il codice dell'applicazione, presentato di seguito, istanzia un ProxyFileHandler con l'indicazione di aprire il file "Files/Secret.txt" e fa delle invocazioni ai diversi metodi. Si noti che particolarmente prima fa una invocazione al metodo di restituzione del nome del file, e dopo fa una doppia invocazione al metodo di restituzione del contenuto, poi una doppia invocazione al metodo getLine per ottenere il testo della seconda riga, e finalmente una invocazione allo stesso metodo, ma adesso per la restituzione della quarta riga.

```
public class ProxyExample{
        public static void main(String[] args){
                FileHandler fh=new ProxyFileHandler( "Files/Secret.txt" );
                System.out.println( "** The name of the file is: " );
                System.out.println( fh.getFileName());
                System.out.println( "** The content of the file is: " );
                System.out.println( fh.getContent() );
                System.out.println( "** The content of the file is (again):" );
                System.out.println( fh.getContent() );
                System.out.println( "** The content of line 2 is: " );
                System.out.println( fh.getLine( 2 ) );
                System.out.println( "** The content of line 2 is (again): " );
                System.out.println( fh.getLine( 2 ) );
```

```
C:\Design Patterns\Structural\Proxy>java ProxyExample

(creating a proxy cache)
** The name of the file is:
Files/Secret.txt

** The content of the file is:
(creating a real handler with file content)
One reason for controlling access to an object
is to defer the full cost of its creation and
initialization until we actually need to use it.
Proxy is appicable whenever there is a need for
a versatile or sophisticated reference to an
object than a simple pointer.

** The content of the file is (again):
One reason for controlling access to an object
is to defer the full cost of its creation and
initialization until we actually need to use it.
Proxy is appicable whenever there is a need for
a versatile or sophisticated reference to an
object than a simple pointer.

** The content of line 2 is:
(accessing from real handler)
"is to defer the full cost of its creation and "

** The content of line 2 is (again):
(accessing from proxy cache)
"is to defer the full cost of its creation and "

** The content of line 4 is:
(accessing from real handler)
"Proxy is appicable whenever there is a need for "
```

Figure 6.3: Esecuzione dell'esempio

```
                System.out.println( "** The content of line 4 is: " );
                System.out.println( fh.getLine( 4 ) );
        }
}
```

Osservazioni sull'esempio

Questo esempio dimostra l'utilizzo del proxy pattern in due ambiti diverse: come virtual proxy e come caché proxy. In entrambi casi l'obbiettivo del proxy è ridurre lo spreco di risorse, in particolare, di memoria e di potenza di calcolo. E' questo obbiettivo quello che crea realmente una distinzione tra questo pattern e il Decorator.

Esecuzione dell'esempio

Si offre di seguito i risultati dell'esecuzione del programma di prova. Le stampe dei messaggi di controllo inseriti nel codice rivelano che l'oggetto RealFileHandler è creato soltanto quando viene invocato per prima volta il metodo getContent, intanto che la precedente invocazione al metodo getName viene risolta completamente dal ProxyFileHandler. La successiva invocazione al metodo getLine con parametro 2, è inoltrata dall'oggetto ProxyFileHandler all'oggetto RealFileHandler, la cui risposta viene immagazzinata nella caché del proxy, prima di essere trasmessa al chiamante. In questo modo, la seguente invocazione al metodo getLine con lo stesso valore come parametro, viene risposta direttamente dal proxy. Invece, l'ultima invocazione a getLine, con un valore diverso come parametro, comporta l'invocazione al rispettivo metodo del RealFileHandler.

6.5 Osservazioni sull'implementazione in Java

La Remote Method Invocation (RMI) che consente l'interazione di oggetti Java distribuiti, utilizza il proxy pattern per l'interfacciamento remoto di oggetti. In questo caso i proxy sono chiamati "stub".

Chapter 7

Chain of Responsibility

7.1 Descrizione

Consente di separare il mittente di una richiesta dal destinatario, in modo di consentire a più di un oggetto di gestire la richiesta. Gli oggetti destinatari vengono messi in catena, e la richiesta trasmessa dentro questa catena fino a trovare un oggetto che la gestisca.

7.2 Esempio

Una azienda commerciale deve gestire le richieste di credito dei clienti (CUSTOMERS). Internamente l'azienda si organizza in diversi livelli di responsabilità. Al livello più basso (VENDOR) viene consentito l'approvazione di richieste fino a un importo determinato. Le richieste che superano questo importo vanno gestite da un livello superiore (SALESMANAGER), il quale ha un altro importo massimo da gestire. Richieste che vanno oltre quest'ultimo importo, vanno gestire da un livello più alto ancora (CLIENT ACCOUNT MANAGER). Il problema riguarda la definizione di un meccanismo di inoltro delle richieste, del quale il chiamante non debba conoscere la struttura.

7.3 Descrizione della soluzione offerta dal pattern

Il "Chain of responsibility" pattern propone la costruzione di una catena di oggetti responsabili della gestione delle richieste pervenute dai clienti. Quando un oggetto della catena riceve una richiesta, analizza se corrisponde a lui gestirla, o, altrimenti, inoltrarla al seguente oggetto dentro la catena. In questo modo, gli oggetti che iniziano la richiesta devono soltanto interfacciarsi con l'oggetto più basso della catena di responsabilità.

7.4 Applicazione del Pattern

Partecipanti

- **Handler**: classe CREDITREQUESTHANDLER.
  - Specifica una interfaccia per la gestione delle richieste.
  - n modo opzionale, implementa un riferimento a un oggetto successore.
- **ConcreteHandler**: classi VENDOR, SALESMANAGER e CLIENTACCOUNTMANAGER.

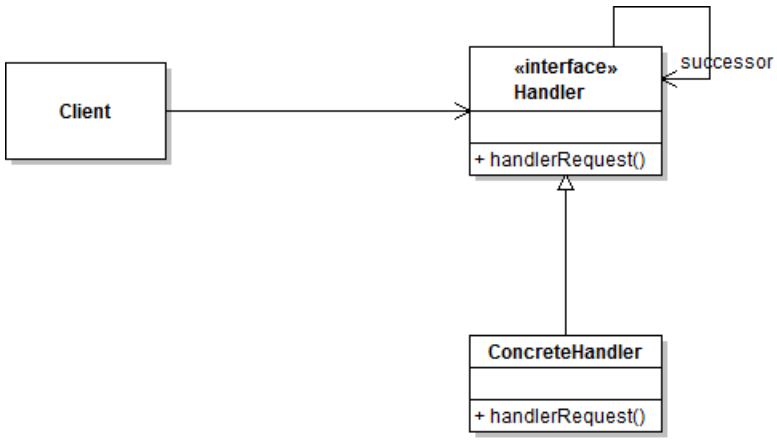


Figure 7.1: Struttura del Pattern

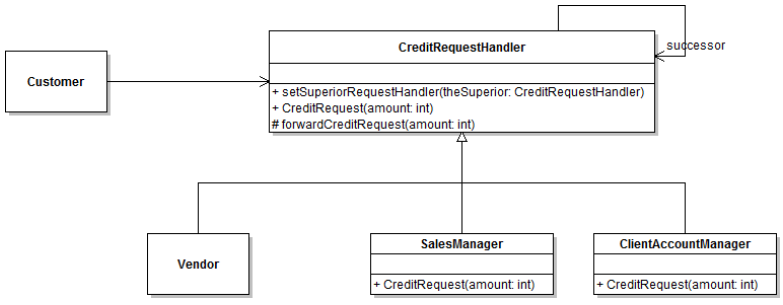


Figure 7.2: Schema del modello

  - Gestiscono le richieste che corrispondono alla propria responsabilità.
  - Accedono ad un successore (se è possibile), nel caso in cui la richiesta non corrisponda alla propria gestione.

- **Client**: classe CUSTOMER.
  - Inoltra una richiesta a un CONCRETEHANDLER della catena.

Descrizione del codice

Si presenta la classe astratta CREDITREQUESTHANDLER che fornisce il meccanismo di costruzione delle catene di oggetti responsabili. Il metodo SETSUPERIORREQUESTHANDLER accetta come parametro un riferimento a un altro CREDITREQUESTHANDLER che viene segnato come responsabile nella catena decisionale. Il metodo creditRequest riceve una richiesta e fornisce come default l'inoltro verso il successivo elemento dentro la catena, tramite la chiamata al metodo FORWARDCREDITREQUEST. Il metodo FORWARDCREDITREQUEST realizza l'inoltro effettivo, soltanto se esiste un riferimento a un seguente elemento dentro la catena, altrimenti solleva una eccezione CREDITREQUESTHANDLEREXCEPTION.

```
public abstract class CreditRequestHandler {
        private CreditRequestHandler successor;

        public void setSuperiorRequestHandler(CreditRequestHandler theSuperior){
                successor = theSuperior;
        }

        public void creditRequest(int amount) throws CreditRequestHandlerException{
                forwardCreditRequest( amount );
        }

        protected void forwardCreditRequest( int amount )
                        throws CreditRequestHandlerException {
                if( successor != null )
                        successor.creditRequest( amount );
                else
                        throw new CreditRequestHandlerException();
        }
}
```

Ogni oggetto responsabile della catena deve estendere la classe precedente, e fornire la particolare implementazione della propria gestione. Nel caso in cui si verifichi la condizione di dover inoltrare la richiesta all'oggetto successivo, deve fare una chiamata al metodo FORWARDCREDITREQUEST ereditato dalla superclasse. Il primo tipo di oggetto responsabile nella catena sarà della classe VENDOR. Gli oggetti VENDOR, in questo esempio, non hanno capacità decisionale, e soltanto il suo ruolo è quello di inoltrare le chiamate al livello successivo. Dato che tutte queste funzioni sono presenti nella superclasse, il VENDOR non richiede codice.

```
public class Vendor extends CreditRequestHandler {}
```

Le classi SALESMANAGER e CLIENTACCOUNTMANAGER estendono CREDITREQUESTMANAGER, specificando la propria gestione nel metodo CREDITREQUEST. Ogni oggetto è in grado di approvare o rifiutare una richiesta di credito fino a un importo predeterminato. L'inoltro delle richieste verso successivi oggetti della catena di responsabilità viene eseguito tramite una chiamata al metodo FORWARDCREDITREQUEST.

```
public class SalesManager extends CreditRequestHandler {
        public void creditRequest(int amount ) throws CreditRequestHandlerException{
                if( amount <= 1000 )
                        if( Math.random() < .3 )
                                System.out.println("Accepted by Sales Manager.");
                        else
                                System.out.println("Not accepted by Sales Manager.");
                else
                        forwardCreditRequest( amount );
        }
}

public class ClientAccountManager extends CreditRequestHandler {
        public void creditRequest( int amount )
                        throws CreditRequestHandlerException {
                if( amount <= 2000 )
                        if( Math.random() < .2 )
                                System.out.println("Accepted by Client Account Manager.");
                        else
                                System.out.println("Not accepted by Client Account Manager.
                else
                        forwardCreditRequest( amount );
        }
}
```

La classe CreditRequestHandlerException serve per rappresentare le eccezioni riguardanti chiamate non gestite da nessun oggetto dell'intera catena:

```
public class CreditRequestHandlerException extends Exception {
        public CreditRequestHandlerException() {
                super( "No handler found to forward the request." );
        }
}
```

La classe Customer rappresenta l'utente della catena di responsabilità. Il metodo requestCredit riceve come uno dei suoi parametri il riferimento all'oggetto della catena a chi deve fare la richiesta.

```
public class Customer {
        public void requestCredit( CreditRequestHandler crHandler, int amount )
                        throws CreditRequestHandlerException {
                crHandler.creditRequest( amount );
        }
}
```

Di seguito si presenta il codice del programma che crea una catena di responsabilità, avendo come elemento iniziale un oggetto Vendor, il quale ha come successore un oggetto SalesManager, il quale, a sua volta, ha come successore un oggetto ClientAccountManager. Si noti che il programma fa richieste di credito per diversi importi, in modo di sfruttare tutto il potere decisionale della catena di responsabilità.

```
public class ChainOfResponsibilityExample {
        public static void main(String[] arg) throws CreditRequestHandlerException{
                ClientAccountManager clientAccountMgr = new ClientAccountManager();
```

```
C:\Design Patterns\Behavioral\Chain of responsibility>java
ChainOfResponsibilityExample

Credit request for : $500
Accepted by Sales Manager.

Credit request for : $1000
Not accepted by Sales Manager.

Credit request for : $1500
Not accepted by Client Account Manager.

Credit request for : $2000
Accepted by Client Account Manager.

Credit request for : $2500
Exception in thread "main" CreditRequestHandlerException: No handler
found to forward the request.
        at
CreditRequestHandler.forwardCreditRequest(CreditRequestHandler.java:17)
        at
ClientAccountManager.creditRequest(ClientAccountManager.java:10)
        at
CreditRequestHandler.forwardCreditRequest(CreditRequestHandler.java:15)
        at SalesManager.creditRequest(SalesManager.java:11)
        at
CreditRequestHandler.forwardCreditRequest(CreditRequestHandler.java:15)
        at
CreditRequestHandler.creditRequest(CreditRequestHandler.java:10)
        at Customer.requestCredit(Customer.java:4)
        at
ChainOfResponsibilityExample.main(ChainOfResponsibilityExample.java:15)
```

Figure 7.3: Esecuzione dell'esempio

```
                SalesManager salesMgr = new SalesManager();
                Vendor vendor = new Vendor();
                vendor.setSuperiorRequestHandler( salesMgr );
                salesMgr.setSuperiorRequestHandler( clientAccountMgr );
                Customer customer = new Customer();
                int i=500;
                while( i <= 2500 ) {
                        System.out.println( "Credit request for : $"+ i );
                        customer.requestCredit( vendor, i );
                        i += 500;
                }
        }
}
```

Osservazioni sull'esempio

Si noti che in questo esempio l'ultima richiesta che il cliente riesce a inoltrare non risulta gestita da nessun oggetto della catena, provocando il sollevamento di una eccezione.

7.5 Osservazioni sull'implementazione in Java

Un altro approccio nell'utilizzo di questo design pattern può osservarsi nella nel meccanismo di Java per la gestione delle eccezioni: ogni volta che viene sollevata una eccezione, questa può essere gestita nello stesso metodo in cui si presenta (tramite le istruzioni "TRY...CATCH"), oppure essere lanciata verso il metodo precedente nello stack di chiamate. A sua volta, questo metodo potrebbe gestire l'eccezione oppure continuare a lanciarlo al successivo. Finalmente, se il metodo MAIN non gestisce l'eccezione, la

Java Virtual Machine ne tiene cura di esso interrompendo l'esecuzione del programma e stampando le informazioni riguardanti l'eccezione.

Chapter 8

Iterator

8.1 Descrizione

Fornisce un modo di accedere sequenzialmente agli oggetti presenti in una collezione, senza esporre la rappresentazione interna di questa.

8.2 Esempio

l percorso di un viaggiatore è rappresentato come una collezione ordinata di oggetti, dove ogni oggetto rappresenta un luogo visitato. La collezione può essere implementata in base a *array*, una *linked list* o qualunque altra struttura. Una applicazione sarebbe interessata in poter accedere agli elementi di questa collezione in una sequenza particolare, ma senza dover interagire direttamente con il tipo di struttura interna.

8.3 Descrizione della soluzione offerta dal pattern

Il pattern "*Iterator*" suggerisce l'implementazione di un oggetto che consenta l'acceso e percorso della collezione, e che fornisca una interfaccia standard verso chi è interessato a percorrerla e ad accede agli elementi.

8.4 Applicazione del Pattern

Partecipanti

- **Iterator**: interfaccia ListIterator.
  - Specifica l'interfaccia per accedere e percorrere la collezione.
- **ConcreteIterator**: oggetto che implementa l'interfaccia ListIterator.
  - Implementa la citata interfaccia.
  - Tiene traccia della posizione corrente all'interno della collezione.
- **Aggregate**: interfaccia List.
  - specifica una interfaccia per la creazione di oggetti Iterator.

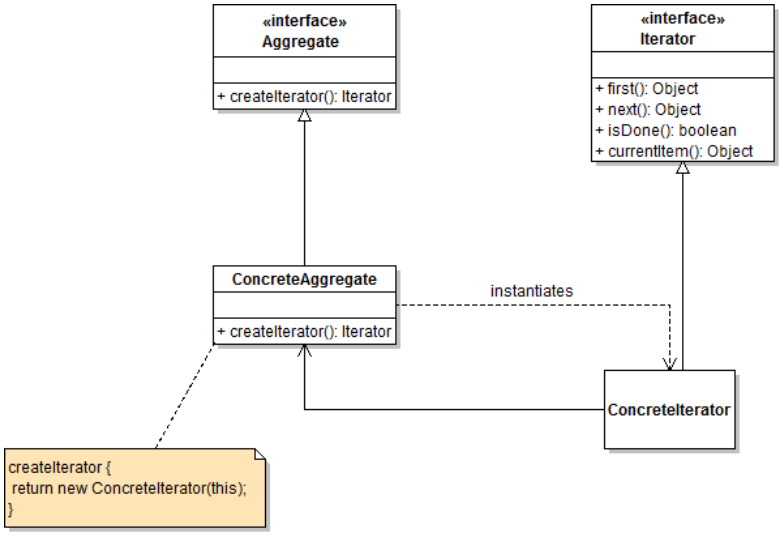


Figure 8.1: Struttura del Pattern

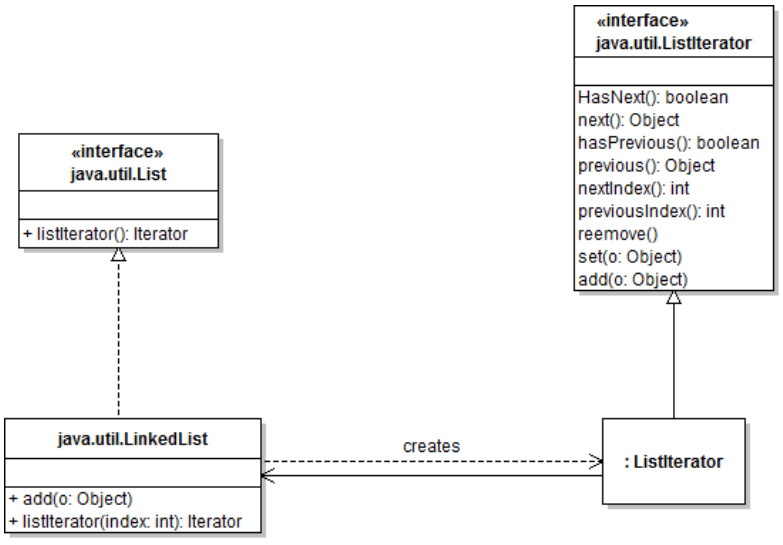


Figure 8.2: Schema del modello

- **ConcreteAggregate**: classe LinkedList.
  - Crea e restituisce una istanza di iterator.

Descrizione del codice

Java fornisce l'implementazione degli **Iterator** per le collezioni che implementano diretta o indirettamente l'interfaccia java.util.List, dove si specifica il metodo listIterator per la restituzione di uno dei tipi di iterator della lista. Esistono tre interfacce di iteraror nelle API di Java:

- java.util.Enumeration: è l'interfaccia dell'**iterator** più semplice, che fornisce soltanto i metodi hasMoreElements() (per determinare se ci sono più elementi) e nextElement() (per spostarsi all'elemento successivo). La classe java.util.StringTokenizer utilizza un oggetto che implementa questa interfaccia per l'identificazione dei diversi tokens di una stringa.
- java.util.Iterator: replica la funzionalità dell'interfaccia Enumeration, con i metodi hasNext() e next(), e aggiunge il metodo remove() per l'eliminazione dell'elemento corrente.
- java.util.ListIterator: estende l'interfaccia Iterator, aggiungendo i metodi hasPrevius() (per determinare se ci sono elementi precedenti quello corrente), previus() (per spostarsi all'elemento precedente, nextIndex() e previousIndex() (per conoscere l'indice dell'elemento seguente e di quello prcedente), setObject(Object o) per settare l'oggetto fornito come parametro nella posizione corrente, e addObject(Object o) per aggiungere un elemento alla collezione.

Il codice svolto per l'esemplificazione di questo pattern utilizza una LinkedList come collezione, e un ListIterator per percorrerla. In una prima fase viene creato il percorso di andata del viaggiatore. Poi viene richiesto alla collezione la restituzione di un ListIterator che servirà a percorrere il cammino di andata e di ritorno.

```
import java.util.LinkedList;
import java.util.ListIterator;

public class IteratorExample {
        public static void main( String[] arg ) {
                LinkedList tour = new LinkedList();
                tour.add( "Santiago" );
                tour.add( "Buenos Aires" );
                tour.add( "Atlanta" );
                tour.add( "New York" );
                tour.add( "Madrid" );
                tour.add( "Torino" );
                tour.add( "Napoli" );
                ListIterator travel = tour.listIterator();
                System.out.println( "Percorso andata" );
                while( travel.hasNext() )
                        System.out.println( ((String) travel.next()) );
                System.out.println( "Percorso ritorno" );
                while( travel.hasPrevious() )
                        System.out.println( ((String) travel.previous()) );
        }
}
```

```
C:\Design Patterns\Behavioral\Iterator >java IteratorExample

Percorso andata
Santiago
Buenos Aires
Atlanta
New York
Madrid
Torino
Napoli

Percorso ritorno
Napoli
Torino
Madrid
New York
Atlanta
Buenos Aires
Santiago
```

Figure 8.3: Esecuzione dell'esempio

Osservazioni sull'implementazione in Java

In questo esempio vengono presentate soltanto le funzioni che l'iterator fornisce per percorrere una collezione.

8.5 Osservazioni sull'implementazione in Java

In una situazione di acceso concorrente ad una collezione, diventa necessario fornire adeguati meccanismi di sincronizzazione per l'iterazione su di essa, come si spiega nel Java Tutorial.

Chapter 9

Observer

9.1 Descrizione

Consente la definizione di associazioni di dipendenza di molti oggetti verso di uno, in modo che se quest'ultimo cambia il suo stato, tutti gli altri sono notificati e aggiornati automaticamente.

9.2 Esempio

Ad un oggetto (Subject) vengono comunicati diversi numeri. Questo oggetto decide in modo casuale di cambiare il suo stato interno, memorizzando il numero ad esso proposto. Altri due oggetti incaricati del monitoraggio dell'oggetto descritto (un Watcher e un Psychologist), devono avere notizie di ogni suo singolo cambio di stato, per eseguire i propri processi di analisi. Il problema è trovare un modo nel quale gli eventi dell'oggetto di riferimento, siano comunicati a tutti gli altri interessati.

9.3 Descrizione della soluzione offerta dal pattern

Il pattern "Observer" assegna all"oggetto monitorato (Subject) il ruolo di registrare ai suoi interni un riferimento agli altri oggetti che devono essere avvisati (ConcreteObservers) degli eventi del Subject, e notificarli tramite l'invocazione a un loro metodo, presente nella interfaccia che devono implementare (Observer). Questo pattern è stato proposto originalmente dai GoF per la manutenzione replicata dello stato del ConcreteSubject nei ConcreteObserver, motivo per il quale sono previsti una copia dello stato del primo nei secondo, e la esistenza di un riferimento del ConcreteSubject nel ConcreteObserver. Nonostante lo espresso nel parafo precedente, si deve tenere in conto che questo modello può servire anche a comunicare eventi, in situazioni nelle quali non sia di interesse gestire una copia dello stato del Subject. Da un'altra parte si noti che non è necessario che ogni ConcreteObserver abbia un riferimento al Subject di interesse, oppure, che i riferimenti siano verso un unico Subject. Le Java API offrono un modello esteso in funzionalità, allineato in questa direzione, che sarà l'approccio utilizzato in questo esempio. Un'altra versione del pattern Observer, esteso con una gestione più completa degli eventi, è implementato dentro l'ambiente di sviluppo G++[16]. In questo modello è consentita la registrazione del Observer presso il Subject, indicando addizionalmente il tipo di evento davanti al quale l'Observer deve essere notificato, e la funzione del Observer da invocare.

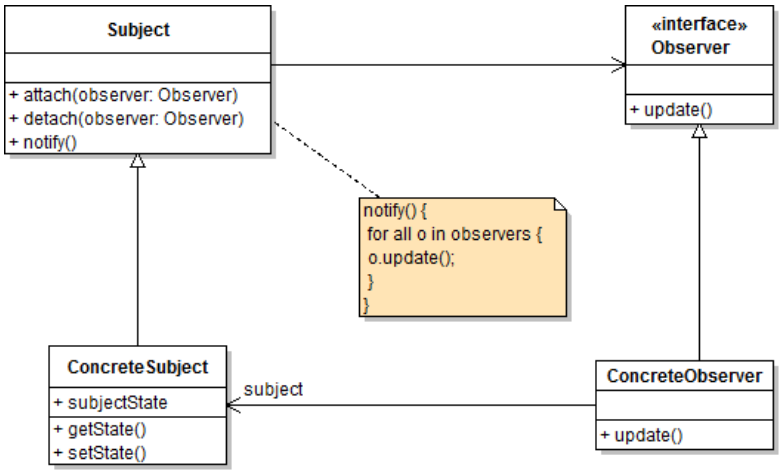


Figure 9.1: Struttura del Pattern

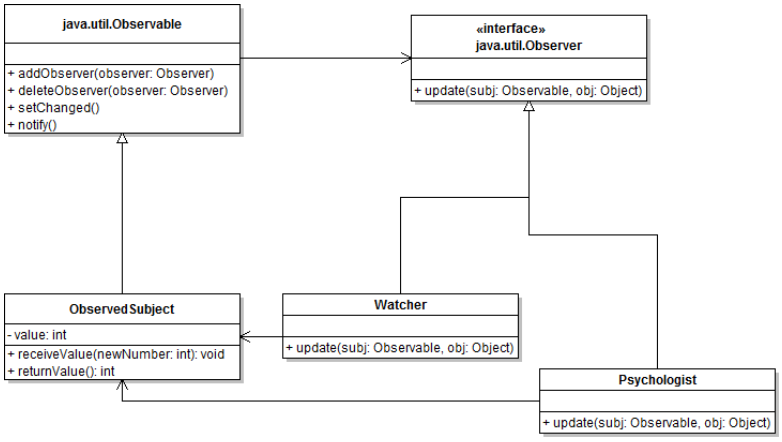


Figure 9.2: Schema del modello

9.4 Applicazione del Pattern

Partecipanti

- **Subject**: classe OBSERVABLE.
  - Ha conoscenza dei propri **Observer**, i quali possono esserci in numero illimitato.
  - Fornisce operazioni per l'addizione e cancellazione di oggetti **Observer**.
  - Fornisce operazioni per la notifica agli **Observer**.
- **Observer**: interfaccia OBSERVER.
  - Specifica una interfaccia per la notifica di eventi agli oggetti interessati in un **Subject**.
- **ConcreteSubject**: classe OBSERVEDSUBJECT.
  - Possiede uno stato dell'interesse dei **ConcreteSubject**.
  - Invoca le operazioni di notifica ereditate dal **Subject**, quando devono essere informati i **ConcreteObserver**.
- **ConcreteObserver**: classi WATCHER e PSYCHOLOGIST.
  - Implementa l'operazione di aggiornamento dell'Observer.

Descrizione del codice

Si presenta di seguito l'applicazione di questo pattern utilizzando come costruttori di base quelli forniti dalle Java API, che sono l'interfaccia JAVA.UTIL.OBSERVER, e la classe JAVA.UTIL.OBSERVABLE:

- Interfaccia JAVA.UTIL.OBSERVER: specifica l'interfaccia che devono implementare i **ConcreteObserver**. Specifica il singolo metodo:
  - VOID UPDATE(OBSERVABLE O, OBJECT ARG): è il metodo che viene chiamato ogni volta che il **Subject** notifica un evento o cambiamento nel proprio stato i **ConcreteObserver**. Al momento dell'invocazione il **Subject** passa come primo parametro un riferimento a sé stesso, e come secondo parametro un oggetto OBJECT qualunque (utile a trasferire informazioni aggiuntive all'Observer)
- classe JAVA.UTIL.OBSERVABLE: serve come classe di base per l'implementazione dei **Subject**. Ha questo costruttore:
  - OBSERVABLE() Costruisce un Observable senza Observers registrati.

  E questi metodi d'interesse:
  - VOID ADDOBSERVER(OBSERVER O): registra l'**Observer** nel suo elenco interno di oggetti da notificare.
  - PROTECTED VOID SETCHANGED(): segna sé stesso come "cambiato", in modo che il metodo HASCHANGED restituisce TRUE.
  - BOOLEAN HASCHANGED(): restituisce true se l'oggetto stesso è stato segnato come "cambiato".

- VOID NOTIFYOBSERVERS(): se l'oggetto è stato segnato come "cambiato", come indicato dal metodo HASCHANGED, fa una notifica a tutti gli **Observer** e poi chiama il metodo CLEARCHANGED per segnare che l'oggetto è adesso nello stato "non cambiato".
- VOID NOTIFYOBSERVERS(OBJECT ARG): agisce in modo simile al metodo precedentemente descritto, ma riceve l'OBJECT ARG come argomento, che è inviato ad ogni **Observer**, come secondo parametro del metodo UPDATE.
- PROTECTED VOID CLEARCHANGED(): indica che l'oggetto non è cambiato, o che la notifica è già stata fatta agli **Observer**. Dopo l'invocazione a questo metodo, HASCHANGED restituisce FALSE.
- INT COUNTOBSERVERS(): restituisce il numero di **Observer** registrati.
- VOID DELETEOBSERVERS():cancella l'elenco degli **Observer** registrati.
- VOID DELETEOBSERVERS(OBSERVER O): cancella un particolare **Observer** dall'elenco dei registrati.

In questo esempio si implementa la classe OBSERVEDSUBJECT **(ConcreteSubject)** che estende la classe OBSERVABLE **(Subject)**. Ogni istanza di questo **ConcreteSubject** riceve semplicemente un numero tramite il metodo RECEIVEVALUE, e con una scelta a caso decide di copiare o meno questo valore nella propria variabile di stato VALUE. Dato che possono esserci **Observer** interessati a monitorare questo cambiamento di stato, quando accadde ciò, viene invocato SETCHANGED per abilitare la procedura di notifica.

```
import java.util.Observer;
import java.util.Observable;

public class ObservedSubject extends Observable {
        private int value = 0;

        public void receiveValue( int newNumber ) {
                if (Math.random() < .5) {
                        System.out.println( "Subject : I like it, I've changed my "
                                + "internal value." );
                        value = newNumber;
                        this.setChanged();
                } else
                        System.out.println( "Subject : I have a number "+ value +
                                " now, and I not interested in the number "
                                + newNumber + "." );
                        this.notifyObservers();
                }

        public int returnValue() {
                return value;
        }
}
```

Si noti che l'istruzione NOTIFYOBSERVERS viene invocata, indipendentemente se si è registrato un cambio di stato nell'oggetto. Questo per dimostrare che la notifica agli **Observer** sarà eseguita soltanto se il cambio di stato è stato segnato da una invocazione al SETCHANGED. La classe WATCHER implementa un **ConcreteObserver**, che ad ogni notifica di cambiamento nel **Subject**, invoca un metodo su quest'ultimo per conoscere il nuovo numero archiviato (stato del **Subject**).

```
import java.util.Observer;
import java.util.Observable;

public class Watcher implements Observer {
        private int changes = 0;

        public void update(Observable obs, Object arg) {
                System.out.println( "Watcher : I see that the Subject holds now a "
                                                                + ((ObservedSubject) obs ).re
                changes++;
        }

        public int observedChanges() {
                return changes;
        }
}
```

Un secondo **ConcreteObserver** è rappresentato dalla classe PSYCOLOGIST, che esegue una operazione diversa dal WATCHER nei confronti della stessa notifica:

```
import java.util.Observer;
import java.util.Observable;

public class Psychologist implements Observer {
        private int countLower, countHigher = 0;

        public void update(Observable obs, Object arg) {
                int value = ((ObservedSubject) obs ).returnValue() ;
                if( value <= 5 )
                        countLower++;
                else
                        countHigher++;
        }

        public String opinion() {
                float media;
                if( (countLower + countHigher ) == 0 )
                        return( "The Subject doesn't like changes.");
                else
                        if( countLower > countHigher )
                                return( "The Subject likes little numbers.");
                        else if ( countLower < countHigher )
                                return( "The Subject likes big numbers.");
                        else
                                return( "The Subject likes little numbers and big numbers.");
        }
}
```

Si presenta di seguito l'applicazione che prima crea una istanza Subject e un'altra di ogni tipo di Observer, e registra questi Observers nel Subject, tramite l'invocazione al metodo addObserver che è ereditato da quest'ultimo, dalla classe Observable.

```
public class ObserverExample {
        public static void main (String[] args) {
```

```
C:\Design Patterns\Behavioral\Observer>java ObserverExample

Main    : Do you like the number 1?
Subject : I like it, I've changed my internal value.
Watcher : I see that the Subject holds now a 1.

Main    : Do you like the number 2?
Subject : I have a number 1 now, and I not interested in the number 2.

Main    : Do you like the number 3?
Subject : I like it, I've changed my internal value.
Watcher : I see that the Subject holds now a 3.

Main    : Do you like the number 4?
Subject : I like it, I've changed my internal value.
Watcher : I see that the Subject holds now a 4.

Main    : Do you like the number 5?
Subject : I like it, I've changed my internal value.
Watcher : I see that the Subject holds now a 5.

Main    : Do you like the number 6?
Subject : I like it, I've changed my internal value.
Watcher : I see that the Subject holds now a 6.

Main    : Do you like the number 7?
Subject : I like it, I've changed my internal value.
Watcher : I see that the Subject holds now a 7.

Main    : Do you like the number 8?
Subject : I like it, I've changed my internal value.
Watcher : I see that the Subject holds now a 8.

Main    : Do you like the number 9?
Subject : I have a number 8 now, and I not interested in the number 9.

Main    : Do you like the number 10?
Subject : I have a number 8 now, and I not interested in the number 10

The Subject has changed 7 times the internal value.
The Psychologist opinion is: The Subject likes little numbers.
```

Figure 9.3: Esecuzione dell'esempio

```
                ObservedSubject s = new ObservedSubject() ;
                Watcher o = new Watcher();
                Psychologist p = new Psychologist();
                s.addObserver( o );
                s.addObserver( p );
                for(int i=1;i<=10;i++){
                        System.out.println( "Main : Do you like the number " + i +"?" );
                        s.receiveValue( i );
                }

        System.out.println( "The Subject has changed " +
                o.observedChanges()
                        + " times the internal value.");
        System.out.println("The Psychologist opinion is:" + p.opinion() );
        }
}
```

Osservazioni sull'esempio

In questo esempio non è stato necessario replicare lo stato del **Subject** negli **Observers**. Nel caso di voler farlo, semplicemente basta di aggiungere una apposita variabile nell'**Observer**, e copiare in essa il valore del momento presente nel **Subject** ad ogni notifica eseguita.

9.5 Osservazioni sull'implementazione in Java

Java estende il modello originalmente proposto dai GoF, in modo di poter associare un singolo **Observer** a più **Subject** contemporaneamente. Questo è consentito dal fatto che il metodo di UPDATE dell'OBSERVER riceve come parametro un riferimento al **Subject** che fa la notifica, consentendo al primo di conoscere quale di tutti i **Subject** la ha originato. E' importante anche notare che questo meccanismo non predispone il modello alla sola gestione di una replica dello stato del **Subject** nell'**Observer**, dato che potrebbe anche essere utilizzato, ad esempio, per la sola comunicazione di eventi. Le particolarità da tenere in conto se si vuole utilizzare le risorse fornite da Java sono:

- OBSERVABLE è una classe che, tramite l'estensione, fornisce i metodi di base del **Subject**. Questo vieta la possibilità che il **Subject** possa essere implementato contemporaneamente come una estensione di un'altra classe.
- La notifica viene eseguita nello stesso thread del **Subject**, costituendo un sistema di notifica sincrono (il metodo update di un **Observer** deve finalizzare e restituire il controllo al metodo notify del **Subject**, prima che questo possa continuare a notificare un altro). In un sistema multi-threding potrebbe essere necessario avviare un nuovo thread ogni volta che un update è eseguito.

Altre idee interessanti riguardanti questo pattern sono presenti negli articoli di Lopez[11] e Bishop[2].

Chapter 10

State

10.1 Descrizione

Consente ad un oggetto modificare il suo comportamento quando il suo stato interno cambia.

10.2 Esempio

Si pensi ad un orologio che possiede due pulsanti: MODE e CHANGE. Il primo pulsante serve per settare il modo di operazione di tre modi possibili: "visualizzazione normale", "modifica delle ore" o "modifica dei minuti". Il secondo pulsante, invece, serve per accendere la luce del display, se è in modalità di visualizzazione normale, oppure per incrementare in una unità le ore o i minuti, se è in modalità di modifica di ore o di minuti. Il diagramma di stati (Figura 10.1) serve a rappresentare il comportamento dell'orologio:

In questo esempio, un approccio semplicistico conduce all'implementazione del codice di ogni operazione come una serie di decisioni (Figura 10.2).

Il problema di questo tipo di codice è che si rende più difficile la manutenzione, perché la creazione di nuovi stati comporta la modifica di tutte le operazioni dove essi sono testati. Da un'altra parte non si tiene una visione dello stato, in modo di capire come agisce l'oggetto

(l'orologio in questo caso), a seconda del proprio stato, perché questo comportamento è spezzato dentro l'insieme di operazioni disponibili. Si vuole definire un meccanismo efficiente per gestire i diversi comportamenti che devono avere le operazioni di un oggetto, secondo gli stati in cui si trovi.

10.3 Descrizione della soluzione offerta dal pattern

Il pattern "State" suggerisce incapsulare, all'interno di una classe, il modo particolare in cui le operazioni di un oggetto (Context) vengono svolte quando lo si trova in quello stato. Ogni classe (ConcreteState) rappresenta un singolo stato possibile del Context e implementa una interfaccia comune (State) contenente le operazioni che il Context delega allo stato. L'oggetto Context deve tenere un riferimento al ConcreteState che rappresenta lo stato corrente.

10.4 Applicazione del Pattern

Partecipanti

- **Context**: classe CLOCK.

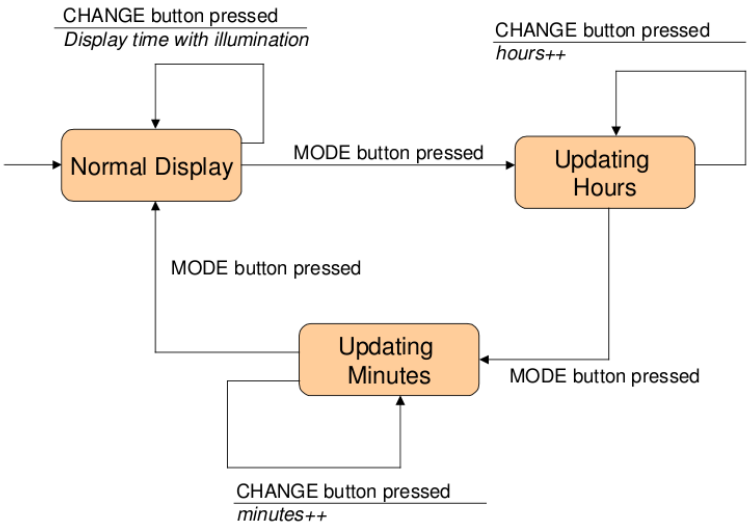


Figure 10.1:

```
operation buttonCHANGEpressed{

   if( clockState = NORMAL_DISPLAY )
      displayTimeWithLight();
   else if( clockState = UPDATING_HOURS )
      hours++;
   else if( clockState = UPDATING_MINUTES )
      minutes++;
   ...

}
```

Figure 10.2:

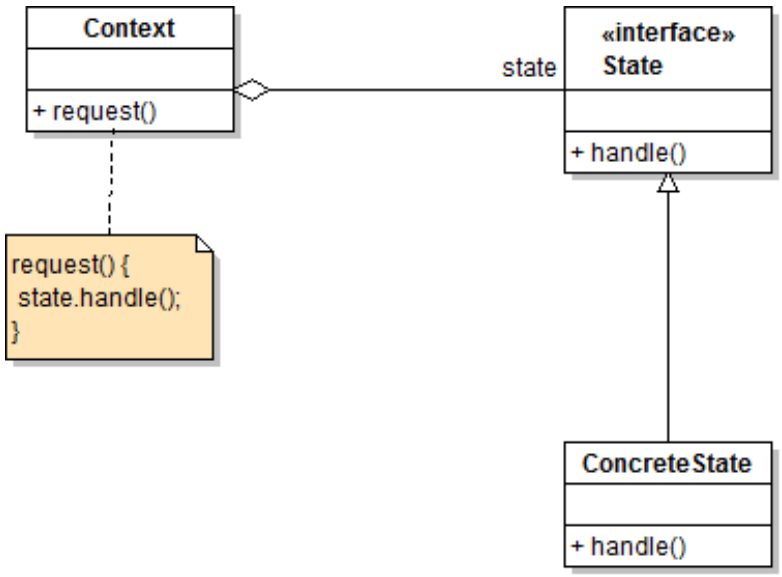


Figure 10.3: Struttura del Pattern

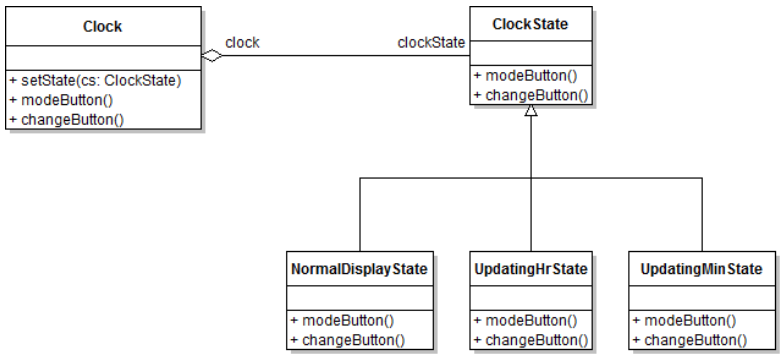


Figure 10.4: Schema del modello

  - Specifica una interfaccia di interesse per i clienti.
  - Mantiene una istanza di **ConcreteState** che rappresenta lo stato corrente

- **State**: classe astratta ClockState.
  - Specifica l'interfaccia delle classi che incapsula il articolare comportamento associato a un particolare stato del **Context**.
- **ConcreteState**: classi NormalDisplayState, UpdatingHrState e UpdatingMinState.
  - Ogni classe implementa il comportamento associato ad uno stato del **Context**.

Descrizione del codice

La classe astratta ClockState **(State)** specifica l'interfaccia che ogni **ConcreteState** deve implementare. Particolarmente questa interfaccia offre due metodi modeButton e changeButton che sono le operazioni da eseguire se viene premuto il tasto MODE o il tasto CHANGE dell'orologio. Queste operazioni hanno comportamenti diversi secondo lo stato in cui ritrova l'orologio. La classe ClockState gestisce anche un riferimento all'oggetto Clock a chi appartiene, in modo che i particolari stati possano accedere alle sue proprietà:

```
public abstract class ClockState {
        protected Clock clock ;

        public ClockState(Clock clock) {
                this.clock = clock;
        }

        public abstract void modeButton();

        public abstract void changeButton();
}
```

Il **ConcreteState** NormalDisplayState estende ClockState. Il suo costruttore richiama il costruttore della superclasse per la gestione del riferimento al rispettivo oggetto Clock. Il metodo modeButton semplicemente cambia lo stato dell'orologio da "visualizzazione normale" a "aggiornamento delle ore" (creando una istanza di UpdatingHrState e associandola allo stato corrente dell'orologio). Il metodo changeButton accende la luce del display per visualizzare l'ora corrente (si ipotizzi che la luce si spegne automaticamente):

```
public class NormalDisplayState extends ClockState {
        public NormalDisplayState(Clock clock) {
                super( clock );
                System.out.println( "** Clock is in normal display.");
        }

        public void modeButton() {
                clock.setState( new UpdatingHrState( clock ) );
        }

        public void changeButton() {
                System.out.print( "LIGHT ON: " );
                clock.showTime();
        }
}
```

La classe UpdatingHrState rappresenta lo stato di modifica del numero delle ore dell'orologio. In questo caso, però, il metodo modeButton cambia lo stato a "modifica dei minuti" (creando una istanza di UpdatingMinState e associandola al Clock). D'altra pare, il metodo changeButton incrementa l'ora corrente in una unità.

```
public class UpdatingHrState extends ClockState {
        public UpdatingHrState(Clock clock) {
                super( clock );
                System.out.println(
        "** UPDATING HR: Press CHANGE button to increase hours.");
        }

        public void modeButton() {
                clock.setState( new UpdatingMinState( clock ) );
        }

        public void changeButton() {
                clock.hr++;
                if(clock.hr == 24)
                        clock.hr = 0;
                System.out.print( "CHANGE pressed - ");
                clock.showTime();
        }
}
```

La classe UpdatingMinState rappresenta lo stato di "modifica dei minuti". In questo stato, il metodo modeButton porta l'orologio allo stato di "visualizzazione normale" (tramite la creazione e associazione all'orologio di una istanza di NormalDisplayState). Il metodo changeButton, invece, incrementa di una unità i minuti dell'orologio.

```
public class UpdatingMinState extends ClockState {
        public UpdatingMinState(Clock clock) {
                super( clock );
                System.out.println(
                        "** UPDATING MIN: Press CHANGE button to increase minutes.");
        }

        public void modeButton() {
```

```
                clock.setState( new NormalDisplayState( clock ) );
        }

        public void changeButton() {
                clock.min++;
                if(clock.min == 60)
                        clock.min = 0;
                System.out.print( "CHANGE pressed - ");
                clock.showTime();
        }
}
```

La classe CLOCK rappresenta il **Context** dello stato. Lo stato corrente di ogni istanza di CLOCK viene gestito con un riferimento verso un oggetto **ConcreteState** (variabile CLOCKSTATE), tramite l'interfaccia CLOCKSTATE. Al momento della creazione, ogni clock viene settato alle ore 12:00 e con nello stato di visualizzazione normale.

```
public class Clock {
        private ClockState clockState;
        public int hr, min;

        public Clock() {
                clockState = new NormalDisplayState( this );
        }

        public void setState( ClockState cs ) {
                clockState = cs;
        }

        public void modeButton() {
                clockState.modeButton();
        }

        public void changeButton() {
                clockState.changeButton();
        }

        public void showTime() {
                System.out.println( "Current time is Hr : " + hr + " Min: "
                        + min );
        }
}
```

Finalmente si presenta il codice dell'applicazione che crea una istanza di Clock ed esegue le seguenti operazioni:

1. Preme per primo il tasto CHANGE: dato che l'orologio è nello stato di visualizzazione normale, dovrebbe mostrare l'ora corrente con la luce del display accesa.

2. Preme il tasto MODE : attiva lo stato di modifica delle ore.

3. Preme due volte il tasto CHANGE: cambia l'ora corrente alle ore 14.

4. Preme il tasto MODE: attiva lo stato di modifica dei minuti

```
C: \Design Patterns\Behavioral\State\Example1>java StateExample

** Clock is in normal display.
LIGHT ON: Current time is Hr : 0 Min: 0

** UPDATING HR: Press CHANGE button to increase hours.
CHANGE pressed - Current time is Hr : 1 Min: 0
CHANGE pressed - Current time is Hr : 2 Min: 0

** UPDATING MIN: Press CHANGE button to increase minutes.
CHANGE pressed - Current time is Hr : 2 Min: 1
CHANGE pressed - Current time is Hr : 2 Min: 2
CHANGE pressed - Current time is Hr : 2 Min: 3
CHANGE pressed - Current time is Hr : 2 Min: 4

** Clock is in normal display.
```

Figure 10.5: Esecuzione dell'esempio

5. Preme quattro volte il tasto CHANGE: cambia il numero dei minuti a 4.

6. Preme il tasto MODE: ritorna allo stato di visualizzazione normale.

```
public class StateExample {
        public static void main ( String arg[] ) {
                Clock theClock = new Clock();
                theClock.changeButton();
                theClock.modeButton();
                theClock.changeButton();
                theClock.changeButton();
                theClock.modeButton();
                theClock.changeButton();
                theClock.changeButton();
                theClock.changeButton();
                theClock.changeButton();
                theClock.modeButton();
        }
}
```

Osservazioni sull'esempio

In questo esempio il cambiamento di stato del **Context** è gestito tramite le stesse operazioni degli stati. Vale dire, una particolare operazione eseguita in uno stato, crea un nuovo stato e lo assegna come stato corrente. Nell'esempio descritto gli stati non vengono riutilizzati, cioè, ogni volta che si cambia di stato, viene creato un nuovo oggetto **ConcreteState**, intanto che il vecchio si perde. Sarebbe più efficiente tenere una singola istanza di ogni **ConcreteState** e assegnare quella corrispondente, tutte le volte che l'orologio cambia stato.

10.5 Osservazioni sull'implementazione in Java

Non ci sono aspetti particolari da tenere in considerazione.

Chapter 11

Strategy

11.1 Descrizione

Consente la definizione di una famiglia d'algoritmi, incapsula ognuno e gli fa intercambiabili fra di loro. Questo permette modificare gli algoritmi in modo indipendente dai clienti che fanno uso di essi.

11.2 Esempio

La progettazione di una applicazione che offre delle funzionalità matematiche, considera la gestione di una apposita classe (MyArray) per la rappresentazione di vettori di numeri. Tra i metodi di questa classe si ha definito uno che esegue la propria stampa. Questo metodo potrebbe stampare il vettore nel seguente modo (chiamato, ad es. MathFormat):
{ 12, -7, 3, ... }
oppure di questo altro modo (chiamato, ad. es. SandardFormat):
Arr[0]=12 Arr[1]=-7 Arr[2]=3 ...
E' anche valido pensare che questi formati potrebbero posteriormente essere sostituiti da altri. Il problema è trovare un modo di isolare l'algoritmo che formatta e stampa il contenuto dell'array, per farlo variare in modo indipendente dal resto dell'implementazione della classe.

11.3 Descrizione della soluzione offerta dal pattern

Lo *"Strategy"* pattern suggerisce l'incapsulazione della logica di ogni particolare algoritmo, in apposite classi (ConcreteStrategy) che implementano l'interfaccia che consente agli oggetti MyArray (Context) di interagire con loro. Questa interfaccia deve fornire un accesso efficiente ai dati del Context, richiesti da ogni ConcreteStrategy, e viceversa.

11.4 Applicazione del Pattern

Partecipanti

- **Strategy**: interfaccia ArrayDisplayFormat.
  - Dichiara una interfaccia comune per tutti gli algoritmi supportati. Il **Context** utilizza questa interfaccia per invocare gli algoritmi definiti in ogni **ConcreteStrategy**.
- **ConcreteStrategy**: classi StandardFormat e MathFormat.

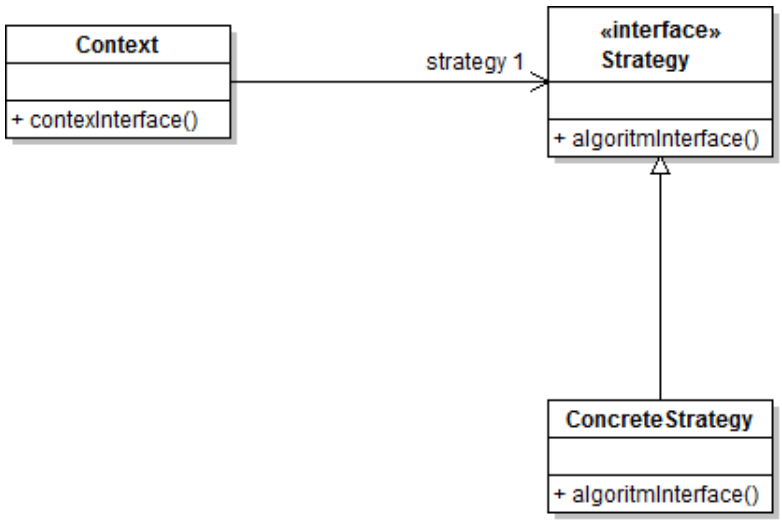


Figure 11.1: Struttura del Pattern

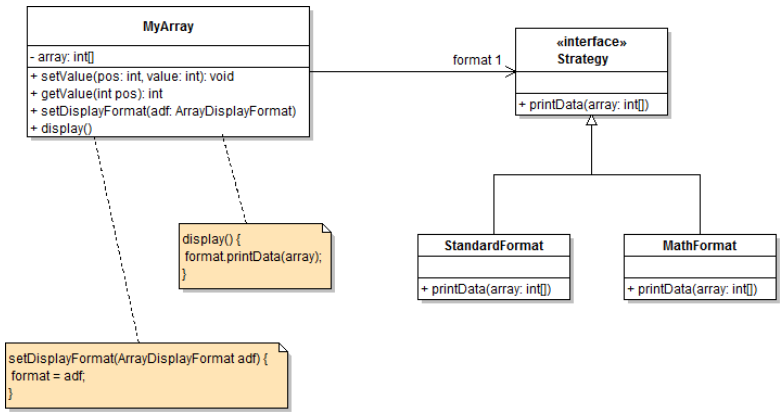


Figure 11.2: Schema del modello

  - Implementano gli algoritmi che usano la interfaccia **Strategy**.

- **Context**: classe MyArray.
  - Viene configurato con un oggetto **ConcreteStrategy** e mantiene un riferimento verso esso.
  - Può specificare una interfaccia che consenta alle **Strategy** accedere ai propri dati.

Descrizione del codice

Si implementa la classe MyArray **(Context)** che mantiene al suo interno un array di numeri, gestiti tramite i metodi setValue e etValue. La particolare modalità di stampa rimane a carico di oggetti che implementano l'interfaccia ArrayDisplayFormat. Il particolare oggetto che incapsula la procedura di stampa scelta, viene settato tramite il metodo setDisplayFormat, intanto che la procedura stessa di stampa viene invocata tramite il metodo display:

```
public class MyArray {
        private int[] array;
        private int size;
        ArrayDisplayFormat format;

        public MyArray( int size ) {
        array = new int[ size ];
        }

        public void setValue( int pos, int value ) {
                array[pos] = value;
        }

        public int getValue( int pos ) {
                return array[pos];
        }

        public int getLength( int pos ) {
                return array.length;
        }

        public void setDisplayFormat( ArrayDisplayFormat adf ) {
                format = adf;
        }

        public void display() {
                format.printData( array );
        }
}
```

Si specifica l'interfaccia ArrayDisplayFormat, da implementare in ogni classe fornitrice dell'operazione di stampa.

```
public interface ArrayDisplayFormat {
        public void printData( int[] arr );
}
```

Le strategie di stampa sono implementate nelle classi StandardFormat (per il formato " a, b, c, ... ") e MathFormat (per il formato "Arr[0]=a Arr[1]=b Arr[2]=c ... "):

```
C: \Design Patterns\Behavioral\Strategy>java StrategyExample

This is the array in 'standard' format :
{ 8, 6, 0, 0, 1, 0, 0, 0, 0, 7 }

This is the array in 'math' format:
Arr[ 0 ] = 8
Arr[ 1 ] = 6
Arr[ 2 ] = 0
Arr[ 3 ] = 0
Arr[ 4 ] = 1
Arr[ 5 ] = 0
Arr[ 6 ] = 0
Arr[ 7 ] = 0
Arr[ 8 ] = 0
Arr[ 9 ] = 7
```

Figure 11.3: Esecuzione dell'esempio

```
public class StandardFormat implements ArrayDisplayFormat {
        public void printData( int[] arr ) {
                System.out.print( "{ " );
                for(int i=0; i < arr.length-1 ; i++ )
                        System.out.print( arr[i] + ", " );
                System.out.println( arr[arr.length-1] + " }" );
        }
}

public class MathFormat implements ArrayDisplayFormat {
        public void printData( int[] arr ) {
                for(int i=0; i < arr.length ; i++ )
                        System.out.println( "Arr[ " + i + " ] = " + arr[i] );
        }
}
```

Finalmente si presenta il codice che fa uso della classe MyArray. Si noti che è questo chi sceglie e istanza la particolare strategia di presentazione dei dati.

```
public class StrategyExample {
        public static void main (String[] arg) {
                MyArray m = new MyArray( 10 );
                m.setValue( 1 , 6 );
                m.setValue( 0 , 8 );
                m.setValue( 4 , 1 );
                m.setValue( 9 , 7 );
                System.out.println("This is the array in 'standard' format");
                m.setDisplayFormat( new StandardFormat() );
                m.display();
                System.out.println("This is the array in 'math' format:");
                m.setDisplayFormat( new MathFormat() );
                m.display();
        }
}
```

Osservazioni sull'esempio

I nomi forniti alle tipologie di stampa sono stati inventati per questo esempio.

11.5 Osservazioni sull'implementazione in Java

Non ci sono aspetti particolari da tenere in conto.

Chapter 12

Visitor

12.1 Descrizione

Rappresenta una operazione da essere eseguita in una collezione di elementi di una struttura. L'operazione può essere modificata senza alterare le classi degli elementi dove opera.

12.2 Esempio

Si consideri una struttura che contiene un insieme eterogeneo di oggetti, su i quali bisogna applicare la stessa operazione, che però è implementata in modo diverso da ogni classe di oggetto. Questa operazione potrebbe semplicemente stampare qualche dato dell'oggetto, formattato in un modo particolare. Per esempio la collezione potrebbe essere un Vector che ha dentro di se oggetti String, Integer, Double o altri Vector. Si noti che se l'oggetto da stampare è un Vector, questo dovrà essere scandito per stampare gli oggetti trovati ai suoi interni. Si consideri anche che l'operazione ad applicare non è in principio implementata negli oggetti appartenenti alla collezione, e che questa operazione potrebbe essere ulteriormente ridefinita. Un approccio possibile sarebbe creare un oggetto con un metodo adeguato per scandire collezioni o stampare i dati dell'oggetto (Figura 12.1)

Questo approccio va bene se si vuole lavorare con pochi tipi di dati, ma intanto questi aumentano il codice diventa una lunga collezione di IF...ELSE. Il problema è trovare un modo di applicare questa operazione a tutti gli oggetti, senza includerla nel codice delle classi degli oggetti.

```
public void printCollection(Collection collection) {
   Iterator iterator = collection.iterator()
   while (iterator.hasNext()) {
      Object o = iterator.next();
      if (o instanceof Collection)
         printCollection( (Collection) o );
      else if ( o instanceof String )
         System.out.println( "'"+o.toString()+"'" );
      else if (o instanceof Float)
         System.out.println( o.toString()+"f" );
      else
         System.out.println( o.toString() );
   }
}
```

Figure 12.1:

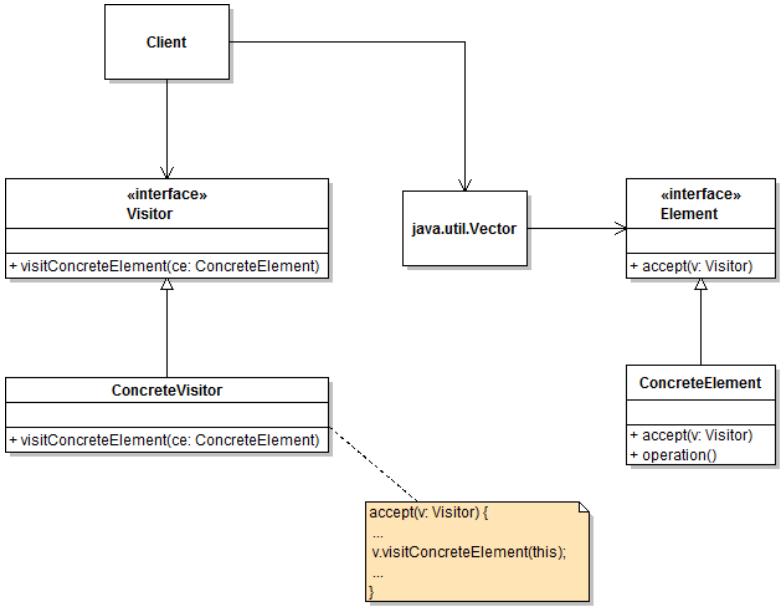


Figure 12.2: Struttura del Pattern

12.3 Descrizione della soluzione offerta dal pattern

La soluzione consiste nella creazione di un oggetto (ConcreteVisitor), che è in grado di percorrere la collezione, e di applicare un metodo proprio su ogni oggetto (Element) visitato nella collezione (avendo un riferimento a questi ultimi come parametro). Per agire in questo modo bisogna fare in modo che ogni oggetto della collezione aderisca ad un'interfaccia (Visitable), che consente al ConcreteVisitor di essere "accettato" da parte di ogni Element. Poi il Visitor, analizzando il tipo di oggetto ricevuto, fa l'invocazione alla particolare operazione che in ogni caso si deve eseguire.

12.4 Applicazione del Pattern

Partecipanti

- **Visitor**: interfaccia VISITOR.
  - Specifica le operazioni di visita per ogni classe di **ConcreteElement**.
- **ConcreteVisitor**: interfaccia VISITOR.
  - Specifica le operazioni di visita per ogni classe di **ConcreteElement**. La firma di ogni operazione identifica la classe che spedisce la richiesta di visita al **ConcreteVisitor**, e in questo modo il visitor determina la concreta classe da visitare. Finalmente il **ConcreteVisitor** accede agli elementi direttamente tramite la sua interfaccia.
- **Element**: interfaccia VISITABLE.
  - Dichiara l'operazione *accept* che riceve un riferimento a un **Visitor** come argomento.
- **ConcreteElement**: classi VISITABLESTRING e VISITABLEFLOAT.
  - Implementa l'interfaccia **Element**.

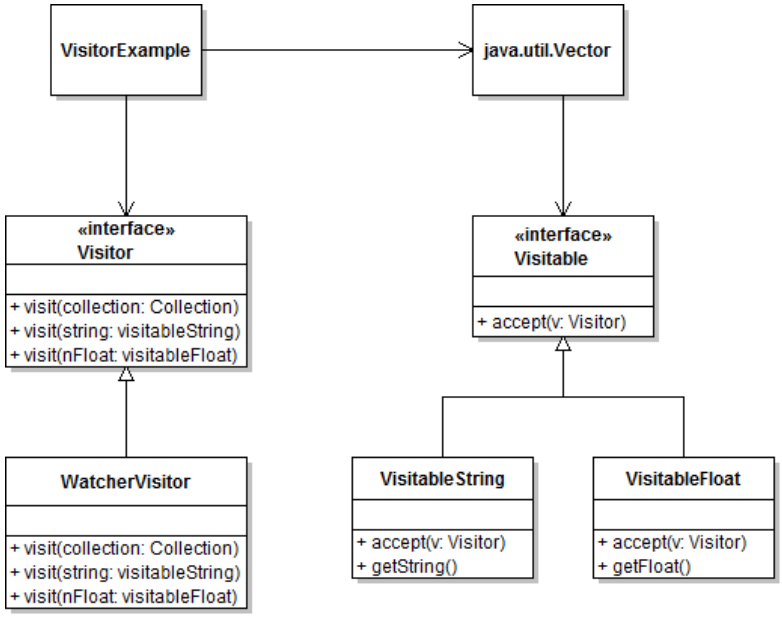


Figure 12.3: Schema del modello

- **ObjectStructure**: classe VECTOR.
  - Offre la possibilità di accettare la visita dei suoi componenti.

Descrizione del codice

Per l'implementazione si definisce l'interfaccia VISITABLE, che dovrà essere implementata da ogni oggetto che accetti la visita di un Visitor:

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

Due concreti oggetti visitabili sono VISITABLESTRING e VISITABLEFLOAT:

```
public class VisitableString implements Visitable {
        private String value;

        public VisitableString(String string) {
                value = string;
        }

        public String getString() {
                return value;
        }

        public void accept( Visitor visitor ) {
                visitor.visit( this );
        }
}
```

```
public class VisitableFloat implements Visitable {
        private Float value;

        public VisitableFloat(float f) {
                value = new Float( f );
        }

        public Float getFloat() {
                return value;
        }

        public void accept( Visitor visitor ) {
                visitor.visit( this );
        }
}
```

Si noti che in entrambi casi, oltre ai metodi particolari di ogni classe, c'è il metodo ACCEPT, dichiarato nell'interfaccia VISITABLE. Questo metodo soltanto riceve un riferimento ad un **Visitor**, e chiama la sua operazione di visita inviando se stesso come riferimento. I **ConcreteVisitor** che sono in grado di scandire la collezione e i suoi oggetti, implementano l'interfaccia VISITOR:

```
import java.util.Collection;

public interface Visitor {
        public void visit( Collection collection );

        public void visit( VisitableString string );

        public void visit( VisitableFloat nFloat );
}
```

Si presenta di seguito il codice della classe WATCHERVISITOR, che corrisponde ad un **Concrete-Visitor**:

```
import java.util.Collection;
import java.util.Iterator;

public class WatcherVisitor implements Visitor {
        public void visit(Collection collection) {
                Iterator iterator = collection.iterator();
                while (iterator.hasNext()) {
                        Object o = iterator.next();
                        if (o instanceof Visitable)
                                ((Visitable)o).accept(this);
                        else if (o instanceof Collection )
                                visit( (Collection) o );
                }
        }

        public void visit(VisitableString vString) {
                System.out.println( "'"+vString.getString()+"'" );
        }

        public void visit(VisitableFloat vFloat) {
```

```
                System.out.println( vFloat.getFloat().toString()+"f" );
        }
}
```

Si noti come il WatcherVisitor isola le operazioni riguardanti ogni tipo di Element in un metodo particolare, essendo un approccio più chiaro rispetto di quello basato su if...else nidificati. Ecco il codice di un'applicazione che gestisce una collezione eterogenea d'oggetti, e applica, tramite il WatcherVisitor, un'operazione di stampa su ogni elemento della collezione. Deve osservarsi che l'ultimo elemento aggiunto alla collezione corrisponde a un Double, che non implementa l'interfaccia Visitable:

```
import java.util.Vector;

        public class VisitorExample {
                public static void main (String[] arg) {
                // Prepare a heterogeneous collection
                Vector untidyObjectCase = new Vector();
                untidyObjectCase.add( new VisitableString( "A string" ) );
                untidyObjectCase.add( new VisitableFloat( 1 ) );
                Vector aVector = new Vector();
                aVector.add( new VisitableString( "Another string" ) );
                aVector.add( new VisitableFloat( 2 ) );
                untidyObjectCase.add( aVector );
                untidyObjectCase.add( new VisitableFloat( 3 ) );
                untidyObjectCase.add( new Double( 4 ) );
                // Visit the collection
                Visitor browser = new WatcherVisitor();
                browser.visit( untidyObjectCase );
        }
}
```

Osservazioni sull'esempio

Si noti che dovuto a che tutti i **ConcreteElement** da visitare implementano il metodo accept allo stesso modo, un'implementazione alternativa potrebbe dichiarare Visitable come una classe astratta che implementa il metodo accept. In questo caso i **ConcreteElement** devono estendere questa classe astratta, ereditando l'accept implementato. Il problema di questo approccio riguarda il fatto che gli Element non potranno ereditare da nessuna altra classe.

Esecuzione dell'esempio

Si noti che l'ultimo elemento della lista (un Double con valore uguale a 4), non implementa l'interfaccia Visitable, ed è trascurato nella scansione della collezione (Figura 12.4).

12.5 Osservazioni sull'implementazione in java

Blosser[3] propone una versione del Visitor in Java, diversa dalla specifica indicata dai GoF, che non richiede fornire nomi diversi per ogni metodo di visita, associato ad una particolare tipologia di oggetto da visitare (esempio, visitString o visitFloat). Tutti possono avere lo steso nome, dato che il metodo particolare da applicare viene identificato discriminato dal tipo di parametro specificato (esempio, visit(VisitableString s) o visit(VisitableFloat f)). Questa caratteristica, insieme

```
C:\Design Patterns\Behavioral\Visitor>java VisitorExample

'A string'
1.0f
'Another string'
2.0f
3.0f
```

Figure 12.4:

alle funzionalità offerte da Java, viene sfruttata nella proposta alternativa del Visitor pattern, che si descrive di seguito.

Implementazione del visitor pattern basata sulla Reflection API

I problemi principali del Visitor Pattern, descritto nell'implementazione precedente sono due:

- Se si vuole aggiungere un nuovo tipo di **ConcreteElement** da visitare, per forza bisogna modificare l'interfaccia Visitor, e quindi, ricompilare tutte le classi coinvolte[1].
- I **ConcreteElement** devono implementare una particolare interfaccia per essere visitati (sebbene questo non è davvero un grosso problema).

Tramite la Reflection API di Java si ottiene un'implementazione più semplice e flessibile, che da soluzione a questi due inconvenienti. Per esemplificare ciò si presenta lo stesso caso dell'esempio anteriore.

Per primo, in questo approccio non è necessario di costringere i **ConcreteElement** da visitare debbano implementare una particolare interfaccia, rendendo innecesaria l'interfaccia VISITABLE. Per questa ragione la collezione da visitare in questo esempio, farà uso soltanto degli oggetti STRING, FLOAT, DOUBLE, e altri forniti da Java.

L'interfaccia che i CONCRETEVISITOR basati nella Reflection devono implementare ha un unico metodo con la firma VISIT(OBJECT O):

```
import java.util.Collection;

public interface ReflectiveVisitor {
        public void visit( Object o );
}
```

Il ConcreteVisitor REFLECTIVEWATCHERVISITOR, che ha la stessa funzionalità del WATCHERVISITOR dell'esempio precedente, implementa l'interfaccia REFLECTIVEVISITOR:

```
import java.util.Vector;
import java.util.Collection;
import java.util.Iterator;
import java.lang.reflect.Method;
import java.lang.reflect.InvocationTargetException;

public class ReflectiveWatcherVisitor implements ReflectiveVisitor {
        public void visit(Collection collection) {
                Iterator iterator = collection.iterator();
                while (iterator.hasNext()) {
```

[1]Si noti che le interfacce in Java "non possono crescere".

```
                        Object o = iterator.next();
                        visit( o );
                }
        }

        public void visit(String vString) {
                System.out.println( "'" + vString + "'" );
        }

        public void visit(Float vFloat) {
                System.out.println( vFloat.toString() + "f" );
        }

        public void defaultVisit (Object o) {
                if (o instanceof Collection )
                        visit( (Collection) o );
                else
                        System.out.println(o.toString());
        }

        public void visit(Object o) {
                try {
                        System.out.println( o.getClass().getName() );
                        Method m = getClass().getMethod( "visit",
                        new Class[] { o.getClass() });
                        m.invoke(this, new Object[] { o });
                } catch (NoSuchMethodException e) {
                        // Do the default implementation
                        defaultVisit(o);
                } catch (IllegalAccessException e) {
                        // Do the default implementation
                        defaultVisit(o);
                } catch (InvocationTargetException e) {
                        // Do the default implementation
                        defaultVisit(o);
                }
        }
}
```

Come già è stato detto, il metodo VISIT(OBJECT O) riceve l'oggetto a visitare, e invoca il metodo VISIT corrispondente al particolare tipo di oggetto. Per fare ciò si prepara una istanza dell'oggetto METHOD che dovrà essere eseguito. La prima istruzione chiave è:

Method m = getClass().getMethod("visit", new Class[] o.getClass());

In questa istruzione viene creato un oggetto METHOD che deve corrispondere ad un metodo della classe del **Visitor**. L'istruzione GETCLASS() restituisce un riferimento ad un oggetto della classe CLASS corrispondente al REFLECTIVEWATCHERVISITOR. Su di questo oggetto viene invocato il metodo GETMETHOD che restituisce un oggetto METHOD, il quale fa riferimento a un metodo appartenente alla classe REFLECTIVEWATCHERVISITOR. Per costruire l'oggetto METHOD bisogna specificare:

- il nome particolare del metodo, che in tutti i casi dell'esempio sarà "VISIT";

- un ARRAY di oggetti della classe CLASS contenenti oggetti della classe Class alla quale appartengono i parametri. In questo caso i metodi VISIT hanno un unico parametro, così che l'array avrà un unico oggetto, corrispondente al tipo del parametro. In questo caso l'array è creato tramite l'istruzione NEW CLASS[] {O.GETCLASS()} (si noti che la variabile o ha un riferimento all'oggetto a trattate.

La seconda istruzione è soltanto l'invocazione al metodo:
m.invoke(this, new Object[] o);
Questa istruzione riceve due parametri, un riferimento all'oggetto attuale, e un array con gli oggetti che corrispondono ai parametri.

Può capitare che il REFLECTIVEWATCHERVISITOR non abbia definito un metodo VISIT per trattare un particolare tipo di oggetto, come è il caso, in questo esempio, degli oggetti DOUBLE. In questo caso viene lanciata una NOSUCHMETHODEXCEPTION (perche il metodo VISIT che riceve il particolare tipo di parametro non è fornito dal REFLECTIVEWATCHERVISITOR), e l'eccezione viene intercettata, per eseguire una DEFAULTVISIT.

Si noti che le collezioni sono gestite prima dal DEFAULTVISIT, perché l'istruzione che costruisce l'oggetto METHOD anche genera una NOSUCHMETHODEXCEPTION quando tenta di costruire un oggetto con un tipo particolare di collezione. Questo significa che non riesce a fare il match tra, per esempio, la visita di un VECTOR (o un'altra particolare collezione, che implementi l'interfaccia COLLECTION) e il metodo VISIT(COLLECTION COLLECTION). Questo match, invece, si riesce nell'invocazione diretta del metodo DEFAULTVISIT, con il previo casting del OBJECT a COLLECTION:
visit((Collection) o);
Ecco il codice dell'esempio:

```
import java.util.Vector;

public class ReflectiveVisitorExample {
        public static void main (String[] arg) {
                // Prepare a heterogeneus collection
                Vector untidyObjectCase = new Vector();
                untidyObjectCase.add( "A string" );
                untidyObjectCase.add( new Float( 1 ) );
                Vector aVector = new Vector();
                aVector.add( "Another string" );
                aVector.add( new Float( 2 ) );
                untidyObjectCase.add( aVector );
                untidyObjectCase.add( new Float( 3 ) );
                untidyObjectCase.add( new Double( 4 ) );
                // Visit the collection
                ReflectiveVisitor browser = new ReflectiveWatcherVisitor();
                browser.visit( untidyObjectCase );
        }
}
```

```
C:\Design Patterns\Behavioral\Visitor>java ReflectiveVisitorExample

'A string'
1.0f
'Another string'
2.0f
3.0f
4.0
```

Figure 12.5: Esecuzione dell'esempio

Bibliography

[1] Bacon David et Al. The “double-checked locking is broken” declaration. URL:http://www.cs.umd.edu/ pugh/java/memoryModel/DoubleCheckedLocking.html;

[2] Bishop Philip. The trick to "Iterator Observers": Factor out common code and make your Iterators observable. URL:http://www.javaworld.com/javaworld/javatips/jw-javatip38.html;

[3] Blosser Jeremy. Reflect on the Visitor design pattern. Implement visitors in Java, using reflection. URL: http://www.javaworld.com/javaworld/javatips/jw-javatip98.html?;

[4] Cooper James W. Java Design Patterns. A tutorial. Upper Saddle River: Addison-Wesley 2000.

[5] Fox Joshua. When is a Singleton not a Singleton? URL:http://www.javaworld.com/javaqa/2000-12/03-qa-1221-singleton.html;

[6] Gamma Erich et Al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.;

[7] Hillside group Web Pages. URL: http://hillside.net/;

[8] Hollub Allen, Programming Java threads in the real world, Part 7. URL: http://www.javaworld.com/javaworld/jw-04-1999/jw-04-toolbox.html;

[9] Horstmann C., Cornell G. Java 2 Tecniche Avanzate. McGraw-Hill, 2000.;

[10] Landini, Ugo. Composite pattern. URL:. http://www.ugolandini.net/;

[11] Lopez Albert. How to decouple the Observer/Observable object model. URL: http://www.javaworld.com/javaworld/javatips/jw-javatip29.html;

[12] Nikander P. Gang of Four Design patterns as UML models. URL: http://www.tml.hut.fi/ pnr/;

[13] Paranj Bala. Learn how to implement the Command pattern in Java. URL: http://www.javaworld.com/javaworld/javatips/jw-javatip68.html.;

[14] Sintes, Tony. The singleton rule. URL: http://www.javaworld.com/javaworld/javaqa/2000-12/03-qa-1221-singleton.html;

[15] Sun. The Java tutorial: a practical guide for programmers. URL: http://java.sun.com/docs/books/tutorial/index.html,;

[16] SYCO. G++ Programming Guide Release 6.2, Italy, May 1998.;

[17] Waldhoff, Rod. Implementing the Singleton Pattern in Java. URL: http://members.tripod.com/rwald/index.html.