Contenuto

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.

[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.

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.

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-thead*, 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 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.

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

1. Abstract Factory

(GoF pag. 87)

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*).

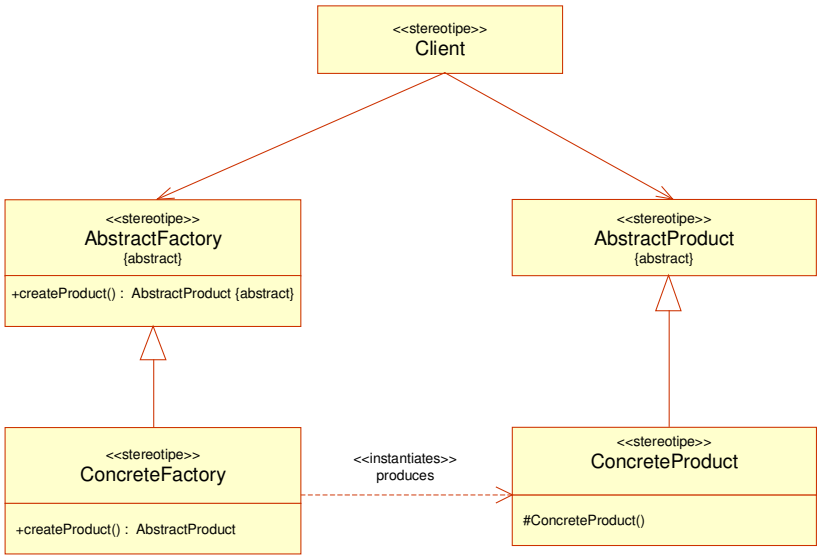
Se si accetta il fatto che questi prodotti offrono agli utenti una stessa interfaccia (cosa non tanto distante dalla realtà), un cliente potrebbe essere in grado di eseguire lo stesso processo di prova su prodotti di entrambe famiglie di prodotti. Ad esempio, potrebbe eseguire prima una registrazione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

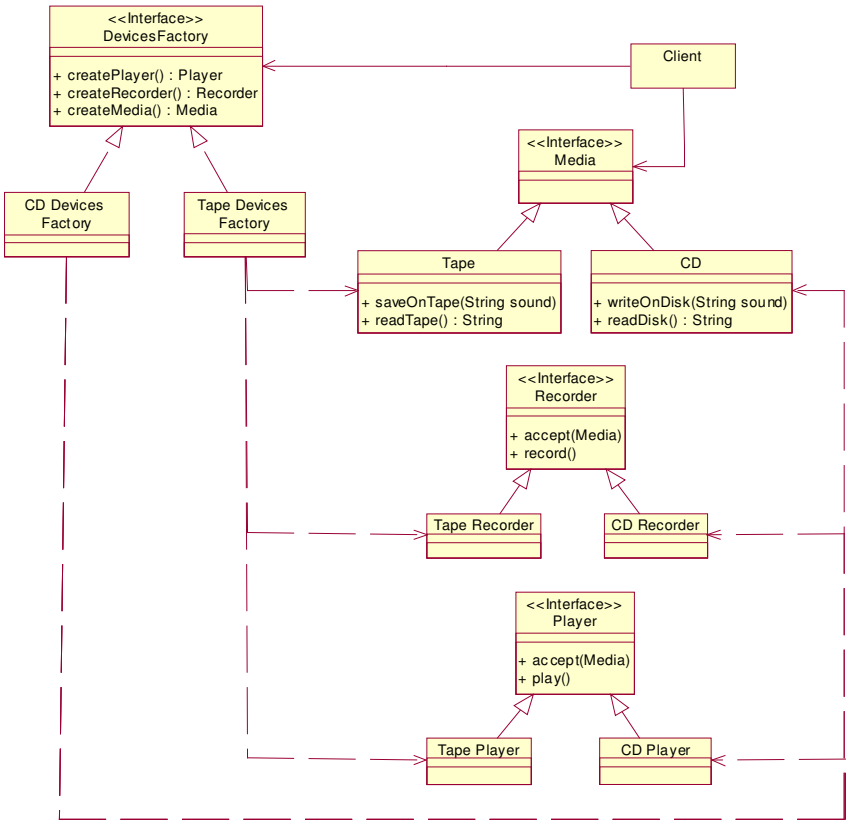
Il pattern "*Abstract Factory*" si basa sulla creazione di interfacce per ogni tipo di prodotto. Ci saranno poi concreti prodotti che implementano queste interfacce, stesse che consentiranno ai clienti di fare uso dei prodotti. Le famiglie di prodotti saranno create da un oggetto noto come *factory*. 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.

1.4. Struttura del pattern



1.5. Applicazione del pattern

Schema del modello



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` è 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() );
    }

}
```

L'interfaccia `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` definisce gli oggetti che utilizzano i prodotti d'ogni 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 d'ogni 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" );
      client.selectTechnology( new CDDevicesFactory() );
      client.test( "Fly me to the moon..." );

   }

}
```

Osservazioni sull'esempio

Nell'esempio si è voluto accennare la stretta interazione tra le classi costituenti ogni famiglia di prodotto, in modo tale di costringere all'utilizzo, 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 l'interfaccia d'ogni tipo di Media verso i corrispondenti registratori e riproduttori varia da una famiglia all'altra.

Esecuzione dell'esempio

```
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...
```

1.6. Osservazioni sull'implementazione in Java

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

2. Builder

(GoF pag. 97)

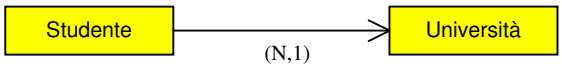
2.1. Descrizione

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, nell'ambito della progettazione di database, lo schema *Entity-Relationship* (ER) è ampiamente utilizzato, anche se non esiste accordo su di un'unica rappresentazione grafica di essi. Per esempio, per costruire un modello informativo nel quale si vuole indicare che "*uno studente appartiene ad una singola università, e un'università può non avere studenti, oppure avere un numero illimitato di questi*", una delle notazioni diffuse consente di rappresentare questo fatto nel seguente modo:

Invece un'altra notazione lo rappresenta così:



Il primo schema, ad effetti di quest'esempio, verrà chiamato "modello non orientato", intanto che il secondo "modello orientato".[5]

Entrambi modelli sono praticamente equipollenti nei confronti di un progetto di sviluppo, essendo possibile notare che l'unica differenza è l'omissione, nella seconda rappresentazione, del nome della relazione esistente tra le due entità[6], e il numero che indica la partecipazione minima d'ogni entità nella relazione.

Uno strumento di supporto alla costruzione di modelli, potrebbe non solo avere la possibilità di 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 quest'esempio si è deciso di avere un modello

[5] Questa è 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 l'interpretazione della partecipazione (card.).

[6] E' certo che se è omesso il nome della relazione soltanto è possibile rappresentare un unico vincolo tra due entità. Il modello presentato si è semplificato soltanto per accennare differenze tra due "prodotti" a costruire, senza rendere più complesso l'esempio.

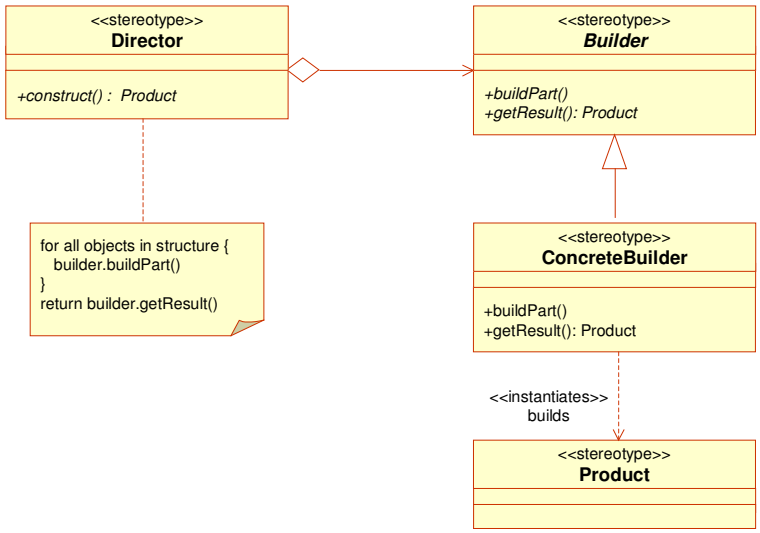
non orientato rappresentato come un'istanza di una classe ERModel, e di un modello orientano rappresentato come una stringa.

Si osservi che siccome c'è una rappresentazione comune di base per entrambi modelli, il processo di costruzione in ogni caso sarà 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

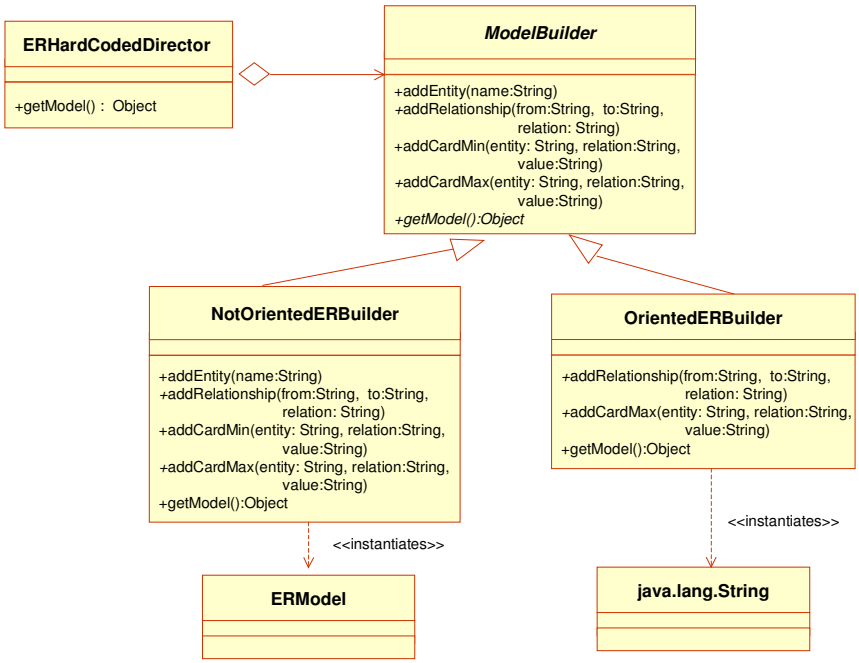
Il "*Builder*" pattern propone separare la "logica del processo di costruzione" dalla "costruzione stessa". Per fare ciò si utilizza un oggetto *Director,* che determina la logica di costruzione del prodotto, e che invia le istruzioni necessarie ad un oggetto *Builder*, incaricato della sua realizzazione. 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 invocherà i metodi del *Builder* scelto secondo il tipo di prodotto desiderato (i *Builder* dovranno implementare un'interfaccia 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. Struttura del Pattern



2.5. Applicazione del pattern

Schema del modello



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.
- **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 essere interessati 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 + ","
                          + 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.

Esecuzione dell'esempio

```
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 ]
```

2.6. 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`.

3. Factory Method

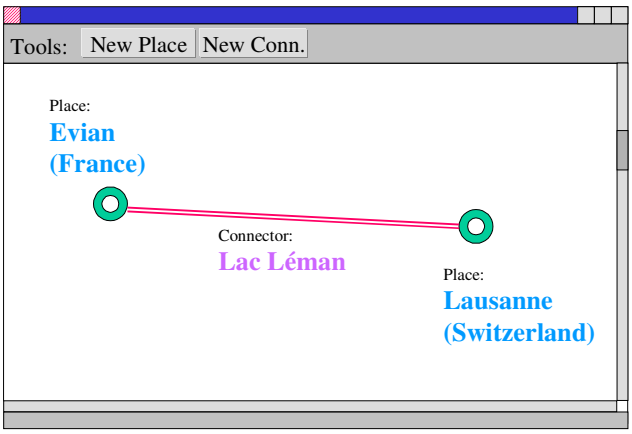
(GoF pag. 107)

3.1. Descrizione

Definisce un'interfaccia per creare oggetti, ma lascia alle sottoclassi la decisione del tipo di classe a istanziare.

3.2. Esempio

Si pensi ad un *framework* per la manipolazione di elementi cartografici. Due classi astratte sono fondamentali per il progetto di questo *framework*: la classe Elemento che rappresenta qualunque tipo di oggetto da posizionare in una mappa (es. luoghi e collegamenti), e la classe Strumento, che fornisce le operazioni comuni di manipolazione degli Elementi. Dato che entrambi classi sono astratte, l'implementazione di un applicativo che sia in grado di gestire un particolare tipo di mappa (che utilizza un insieme definito di Elementi), richiede l'estensione di esse.

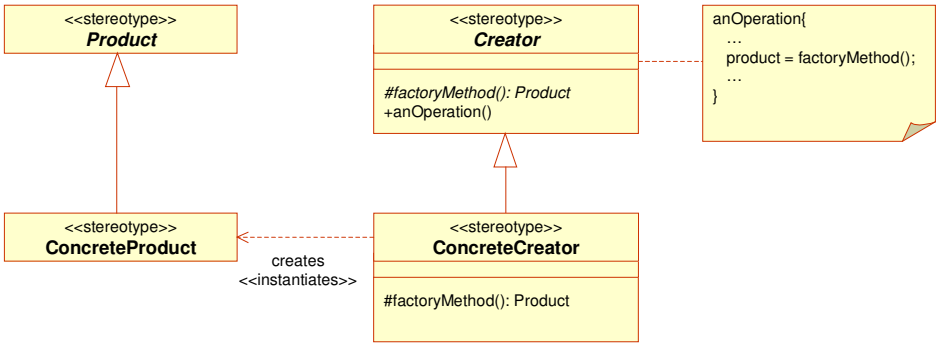


Si fa notare che lo strumento, è in grado di stabilire quando un particolare tipo di elemento deve essere creato (ad esempio, dopo di aver richiesto un identificativo per un nuovo elemento), ma non il tipo particolare di Elemento a creare. In altre parole, il *framework* deve creare istanze di classi, ma soltanto è in grado di avere conoscenza delle classi astratte, che non possono essere istanziate.

3.3. Soluzione

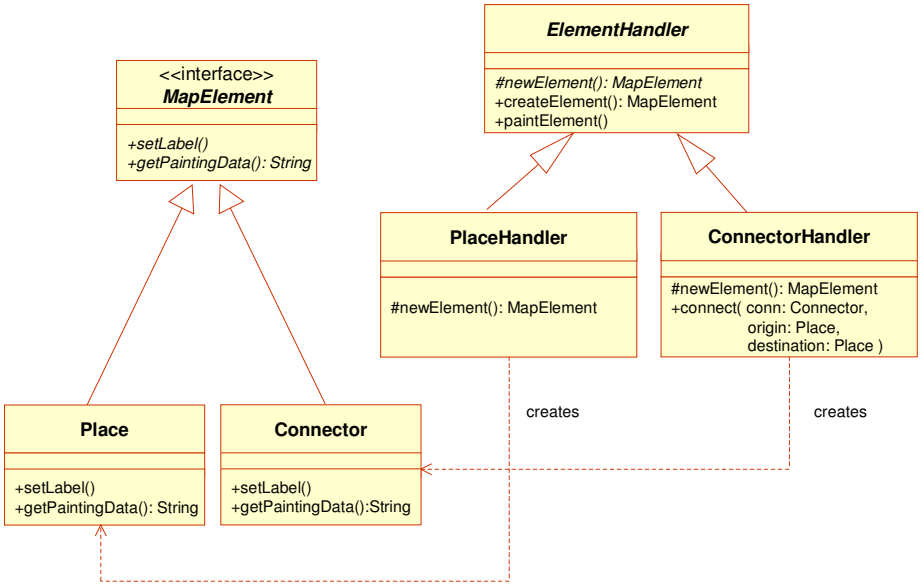
Il pattern "*Factory Method*" suggerisce il portare via dal *framework* la creazione di ogni particolare tipo di Elemento. Per fare ciò, verrà delegato alle sottoclassi dello Strumento, che specializzano le funzioni di gestione di ogni tipo di Elemento, il compito di creare le particolari istanze di classi che siano necessarie.

3.4. Struttura del pattern



3.5. Applicazione del pattern

Schema del modello



Partecipanti

- **Product**: classe astratta `MapElement`.
  - Definisce l'interfaccia di tutti gli elementi da utilizzare nell'applicazione.
- **ConcreteProduct**: classi `Place` e `Connector`
  - Implementano i concreti prodotti.
- **Creator**: classe `ElementHandler`.

  - Dichiara il *factory method* (metodo `newElement`) che restituisce un oggetto della classe **Product**.
  - Richiama il *factory method* per creare i **Product**.
- **ConcreteCreator**: classi `PlaceHandler` e `ConnectorHandler`
  - Redefine il *factory method* per restituire una istanza di `ConcreteProduct`.

Descrizione del codice

L'interfaccia `MapElement` fornisce la definizione dei servizi comuni a tutti i **Product** da gestire nell'applicazione, che nel caso di questo diventano due: `Place` (luoghi) e `Connector` (collegamenti fra luoghi). A livello di *framework*, insieme al `MapElement` esiste l'`ElementHandler`, che fornisce la procedura di creazione e gestione di questi oggetti. L'`ElementHandler` si specializza in un `PlaceHandler` e in un `ConnectorHandler`, che hanno il compito di istanziare il particolare tipo di `MapElement`, quando sia necessario.

Si presenta adesso il codice delle classi appartenenti al *framework*, vale dire, l'interfaccia `MapElement`: e la classe astratta `ElementHandler`:

```
public interface MapElement {

   public abstract void setLabel( String id );

   public abstract String getPaintingData();

}
```

```
import java.io.*;
public abstract class ElementHandler {

   public MapElement createElement( ) throws IOException {
      BufferedReader reader = new BufferedReader( new InputStreamReader(
                                                  System.in) );
      System.out.println( "Enter a label for the element: ");
      String label = reader.readLine();
      MapElement element = newElement( );
      element.setLabel( label );
      return element;
   }

   public abstract MapElement newElement();

   public void paintElement(MapElement element) {
        System.out.println( element.getPaintingData() );
   }

}
```

Si noti che il metodo `createElement` fornisce il codice necessario per creare qualunque tipo di **Product** (`MapElement`). L'istanziazione del particolare elemento è delegata al metodo `newElement`, il quale è dichiarato astratto.

Quando viene sviluppata l'applicazione cartografica a partire di questo *framework*, si implementano le classi concrete per la rappresentazione degli elementi della mappa e per la loro gestione, vale dire i **ConcreteProduct** e i **ConcreteCreator**.

Nel caso dell'esempio gli elementi della mappa (**ConcreteProduct**) corrispondono alle classi `Place` (per i luoghi) e `Connector` (per i collegamenti):

```
class Place implements MapElement {

   private String placeLabel;

   public void setLabel( String label ) {
      placeLabel = label;
   }

   public String getPaintingData() {
      return "city: "+ placeLabel;
   }

}
```

```
class Connector implements MapElement {

   private String connectorLabel;
   Place place1, place2;

   public void setLabel( String label ) {
      connectorLabel = label;
   }

   public void setPlacesConnected( Place origin, Place destination ) {
      place1 = origin;
      place2 = destination;
    }

   public String getPaintingData() {
      return connectorLabel + " [from " +
             place1.getPaintingData() + " to " +
           place2.getPaintingData() + "]";
   }

}
```

Uno dei **ConcreteCreator** corrisponde alla classe `PlaceHandler`, che implementa la creazione di un oggetto `Place` nella chiamata al metodo `newElement`.

```
public class PlaceHandler extends ElementHandler {

  public MapElement newElement() {
     return new Place();
  }

}
```

Allo stesso modo, l'altro **ConcreteCreator**, progettato per la gestione dei **Connector**, corrisponde alla classe `ConnectorHandler`, che anche estende l'**ElementHandler**, e implementa il metodo `newElement`, per creare oggetti della classe `Connector`.

```
public class ConnectorHandler extends ElementHandler {

  public MapElement newElement() {
     return new Connector();
  }

  public void connect(Connector conn, Place origin, Place destination) {
     conn.setPlacesConnected( origin, destination );
  }

}
```

Si è inclusa una funzionalità aggiuntiva nella classe `ConnectorHandler`, che serve a connettere due luoghi (metodo `connect`.

Finalmente si presenta il codice dell'applicazione che dimostra il funzionamento di questo pattern:

```
import java.io.*;
public class FactoryMethodExample {

    public static void main (String[] arg) throws IOException {

       // Creates the tools for handling elements
       ConnectorHandler cTool = new ConnectorHandler();
       PlaceHandler pTool = new PlaceHandler();

       // Vars
       Place startPoint, endPoint;
       Connector route;

       // Creates two places and one connector
       System.out.println( "1st. place creation" );
       startPoint  = (Place) pTool.createElement();
       System.out.println( "2nd. place creation" );
       endPoint    = (Place) pTool.createElement();
       System.out.println( "Connector creation" );
       route       = (Connector) cTool.createElement();

       // Links places with the connection
       cTool.connect( route, startPoint , endPoint );

       // Paints the entire map
       pTool.paintElement( startPoint );
       pTool.paintElement( endPoint );
       cTool.paintElement( route );

    }

}
```

Osservazioni sull'esempio

In questo esempio la gerarchia di creatori (`ElementHandler` – `PlaceHandler` – `ConnectorHandler`) rispecchia la gerarchia di prodotti (`MapElement` – `Place` - `Connector`), ma questo non è un requisito strutturale imposto dal *pattern*: è perfettamente possibile trovarsi davanti a prodotti condivisi tra due o più **ConcreteCreator**.

Esecuzione dell'esempio

```
C:\Design Patterns\Creational\Factory Method>java FactoryMethodExample

1st. place creation
Enter a label for the element:
Evian

2nd. place creation
Enter a label for the element:
Lausanne

Connector creation
Enter a label for the element:
Lac Léman

city: Evian
city: Lausanne
Lac Léman [from city: Evian to city: Lausanne]
```

3.6. Osservazioni sull'implementazione in Java

Si vuole rendere noto che il *factory method* (`newElement`) dichiara come tipo da restituire al punto di chiamata, sia nel **Creator** (`ElementHandler`), sia in ogni **ConcreteCreator** (`PlaceHandler` e `ConnectorHandler`), un oggetto di tipo **Product** (`MapElement`), invece dei particolari tipi da produrre (`Place` e `Connector`). Questo è dovuto al fatto che le sottoclassi che redefiniscono un metodo devono esplicitare lo stesso tipo di ritorno che quello indicato nella dichiarazione del metodo nella superclasse.

4. Prototype

(GoF pag. 117)

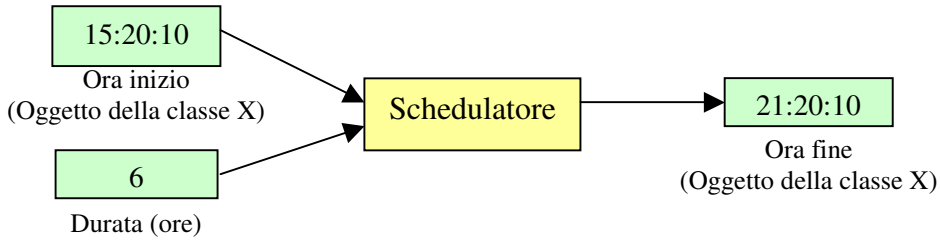
4.1. Descrizione

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.

4.2. 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.

4.3. 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.

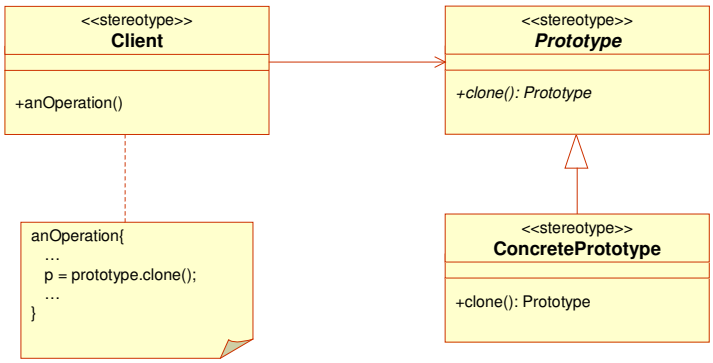
Il Prototype pattern è suggerito per essere applicato nel caso dei sistemi che devono essere indipendenti del modo in quale i loro prodotti sono creati, composti e rappresentati, e

- quando le classi a istanziare sono specificate in run-time; o
- quando si vuole evitare la costruzione di gerarchie di factories parallele alle gerarchie di prodotti; o

- quando le istanze delle classi a istanziare hanno un insieme ridotto di stati possibili.

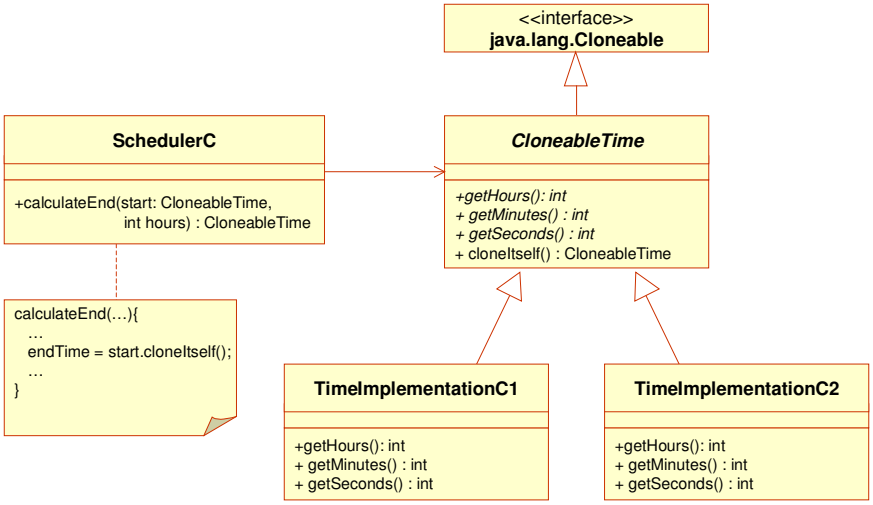
Autori come Cooper [4] hanno adattato questo pattern a Java utilizzando il concetto di clonazione o copia tramite serializzazione, per risolvere il problema che ha dato origine al Pattern. Questo approccio sarà esemplificato di seguito, intanto che un secondo approccio, basato sulla Java Reflection API,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

4.4. Struttura del pattern



4.5. Applicazione del modello

Schema 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 TimeImplementationC1 e TimeImplementationC2
  - Implementano le particolari versioni di oggetti da utilizzare e clonare (estendono CloneableTime).
- **Client:** 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.Cloneable`, e devono invocare il metodo privato `clone` ereditato dalla classe `java.lang.Object`.

Le classi `TimeImplementationC1` e `TimeImplementationC2` che 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;
       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.

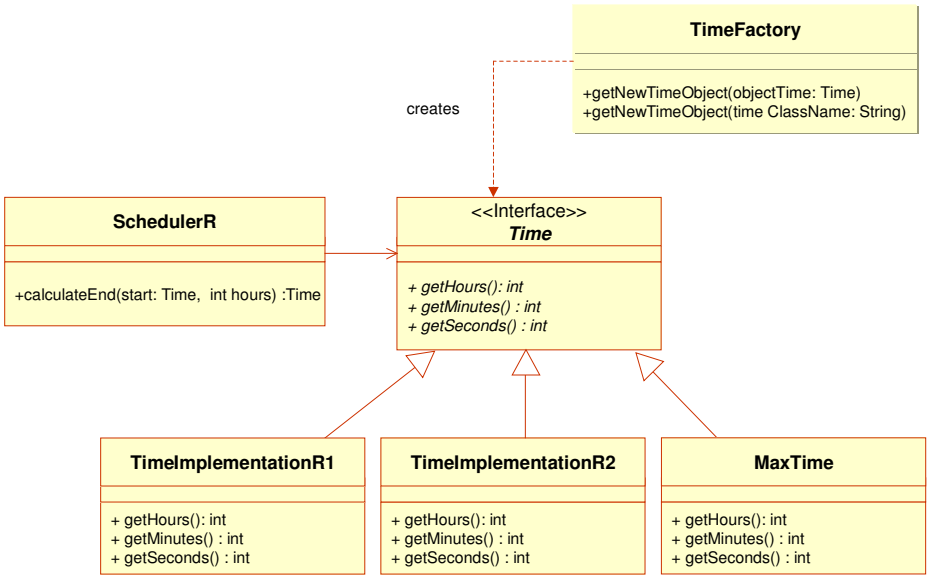
Esecuzione dell'esempio

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

Using TimeImplementationC1:
End: 21:20:10

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

4.6. 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;
	}

	public 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:

```
Class timeClass = objectTime.getClass();
Constructor timeClassConstr = timeClass.getConstructor( new Class[] { });
return (Time) timeClassConstr.newInstance( new Object[]{ } );
```

La procedura di creazione è la seguente[7]:

- 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

[7] 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].

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.

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:

```
Class timeClass = 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() );

      }
}
```

Esecuzione dell'esempio

```
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
```

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.

5. Singleton

(GoF pag. 117)

5.1. Descrizione

Specifica i tipi di oggetti a creare, utilizzando un'istanza prototipa, e crea nuove istanze tramite la copia di questo prototipo.

5.2. Esempio

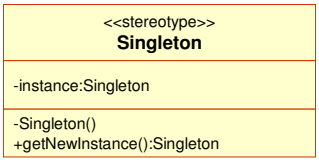
Un applicativo deve istanziare un oggetto che gestisce una stampante. Questo oggetto deve essere unico, vale dire, deve esserci soltanto una sola istanza di esso, altrimenti, potrebbero risultare dei problemi nella gestione della risorsa.

Il problema è la definizione di una classe che garantisca la creazione di un'unica istanza all'interno del programma.

5.3. Descrizione della soluzione offerta dal pattern

Il "*Singleton*" pattern definisce una classe della quale è possibile la istanziazione di un unico oggetto, tramite l'invocazione a un metodo della classe, incaricato della produzione degli oggetti. Le diverse richieste di istanziazione, comportano la restituzione di un riferimento allo stesso oggetto.

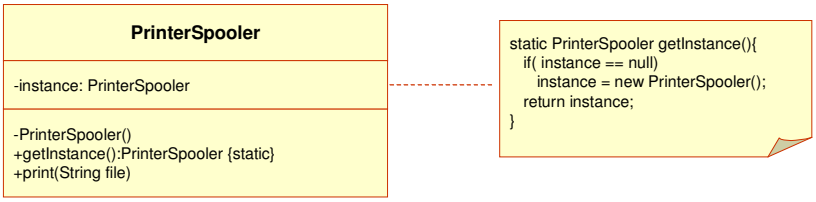
5.4. Struttura del pattern



5.5. Applicazione del modello

Schema del modello

Si presenta di seguito una delle implementazioni più semplici del Singleton:



Partecipanti

- **Singleton**: classe PrinterSpooler.
  - Definisce un metodo getInstance che restituisce un riferimento alla unica istanza di se stessa.
  - E' responsabile della creazione della propria unica istanza.

Descrizione dl codice

Si propongono adesso tre diverse implementazioni del *Singleton* pattern in Java. Si discutono le caratteristiche di ognuna di esse.

a) Singleton come classe statica

E' il modello più semplice, ma che in realtà non è un vero e proprio **Singleton**, perché soltanto si lavora con una classe statica, non un oggetto [14], [17]. Questa classe statica ha metodi statici che offrono i servizi richiesti.

```
public static class PrinterSpooler {

  private PrinterSpooler() {

   }

  public static void print (String msg) {
     System.out.println( msg );
  }

}
```

Si noti che il costruttore è dichiarato privato, per evitare l'istanziazione di oggetti della classe.

L'invocazione all'oggetto sarà una istruzione simile a:

```
PrinterSpooler.print( somethingToPrint );
```

Questa versione di Singleton è chiamato *Booch utility*, perché è stato Grady Booch a identificarlo [8]. Uno dei problemi principali che ha questa versione è la necessità di conoscere a tempo di caricamento delle classi tutta l'informazione necessaria per creare il **Singleton**. Un altro problema è l'impossibilità di implementare interfacce tramite questo approccio.

b) Singleton creato da un metodo statico

Il **Singleton** è implementato come una classe che ha un metodo statico (`getInstance`) che deve essere chiamato per restituire l'istanza del **Singleton**. L'oggetto **Singleton** verrà istanziato solo la prima volta che il metodo sia invocato, in modo che eventuali informazioni necessarie per creare il **Singleton** possano essere fornite in tempo di esecuzione. Le veci successive sarà restituito un riferimento allo stesso oggetto.

```
public class PrinterSpooler {

 private static PrinterSpooler instance;

   private PrinterSpooler() {
```

```
    }

   public static PrinterSpooler getInstance() {
      if ( instance==null) {
         instance = new PrinterSpooler();
      }
      return instance;
   }

  public void print (String msg) {
     System.out.println( msg );
  }

}
```

In questa implementazione il costruttore continua ad essere statico, per costringere all'utilizzo del metodo `getInstance()`, per ricavare gli oggetti:

```
...
PrinterSpooler theUnique = PrinterSpooler.getInstance();
theUnique.print( somethingToPrint );
//if we try this:
PrinterSpooler mayBeOther = PrinterSpooler.getInstance();
//then...
if( theUnique != mayBeOther )   → false!!!
...
```

c) Singleton *multi thread*

La implementazione precedente non va bene nel caso di una esecuzione *multithread*. Si pensi al caso nel quale un *thread* fa l'invocazione, per prima volta, al metodo `getInstance` e dopo di testare se l'istanza è già stata creata con l'istruzione `if( instance == null)`, viene sospeso da un altro *thread* che invoca lo stesso metodo. Dato che il **Singleton** non è ancora creato, questo secondo *thread* potrebbe crearlo. Dopo, quando il controllo è restituito al primo *thread*, questo crea una nuova istanza di Singleton.

La soluzione e sincronizzare il metodo `getInstance`:

```
public class PrinterSpooler {

 private static PrinterSpooler instance;

   private PrinterSpooler() {

   }

   public static synchronized PrinterSpooler getInstance() {
      if ( instance==null) {
         instance = new PrinterSpooler();
      }
      return instance;
   }


  public void print (String msg) {
     System.out.println( msg );
  }

}
```

La soluzione presentata può essere considerata inefficiente perché ogni volta che si fa una invocazione al metodo `getInstance` deve essere

acquisito un *lock*. Una soluzione scorretta in Java è la suggerita da Doug Schmidt e presentata da Holub [8]. Questa soluzione utilizza il lock soltanto se l'istanza non è già stata creata (strategia chiamata "*double-checked locking*"):

```
public class PrinterSpooler {

 private static PrinterSpooler instance;

   private PrinterSpooler() {

   }

   public static PrinterSpooler getInstance() {
      if ( instance == null) {
             synchronized( PrinterSpooler.class ) {
                    if( instance == null )
                           instance = new PrinterSpooler();
             }
      }
      return instance;
   }


  public void print (String msg) {
     System.out.println( msg );
  }

}
```

Bacon et. al. [1] spiegano come i criteri di ottimizzazione, al momento di implementare la Java Virtual Machine, possono far generare più di una istanza al programma presentato.

Osservazioni sull'esempio

Si è voluto presentare un esempio banale, che serve a esemplificare diverse implementazioni di **Singleton**. Data la semplicità dell'esempio, non si presenta l'esecuzione di un *main program*, come nei casi dei pattern precedenti.

5.6. Osservazioni sull'implementazione in Java

Altre considerazioni da studiare al momento di proporre l'implementazione di un *Singeton* sono descritti da Fox [5]. Aspetti da tenere in conto sono:

- Presenza di *Singletons* in multiple *virtual machines.*
- *Singletons* caricati contemporaneamente da diversi *class loaders*.
- *Singletons* distrutti dal *garbage collector*, e dopo ricaricati quando sono necessari.
- Presenza di multiple istanze come sottoclassi di un *Singleton*.
- Copia di *Singletons* come risultato di un doppio processo di deserializzazione.

6. Adapter

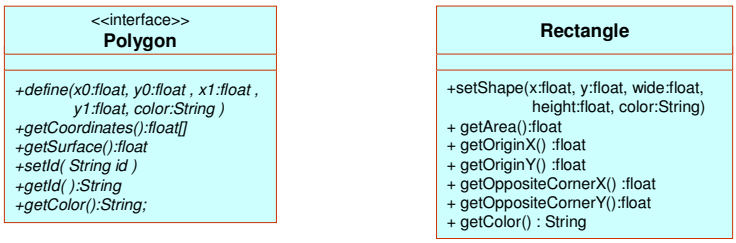
(GoF pag. 141)

6.1. 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.

6.2. 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.



New interface Available class

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

6.3. Descrizione della soluzione offerta dal pattern

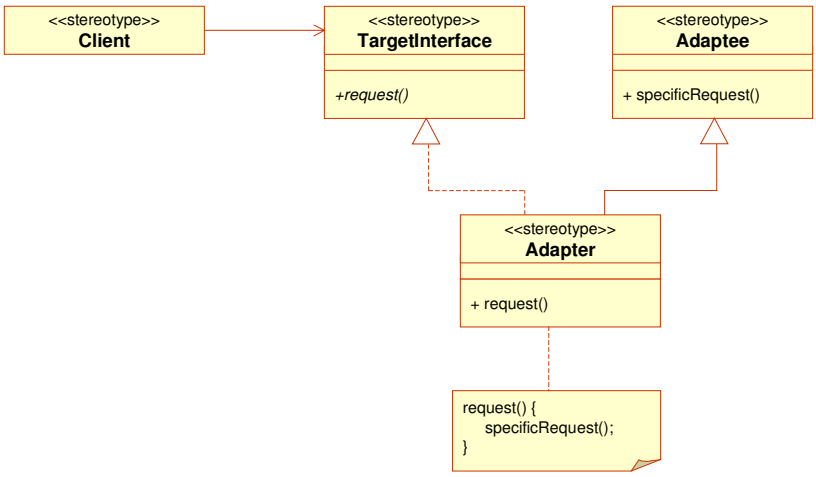
L'*Adapter* pattern offre due soluzioni possibili, denominate *Class Adapter* e *Object Adapter*, che si spiegano di seguito:

- 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.

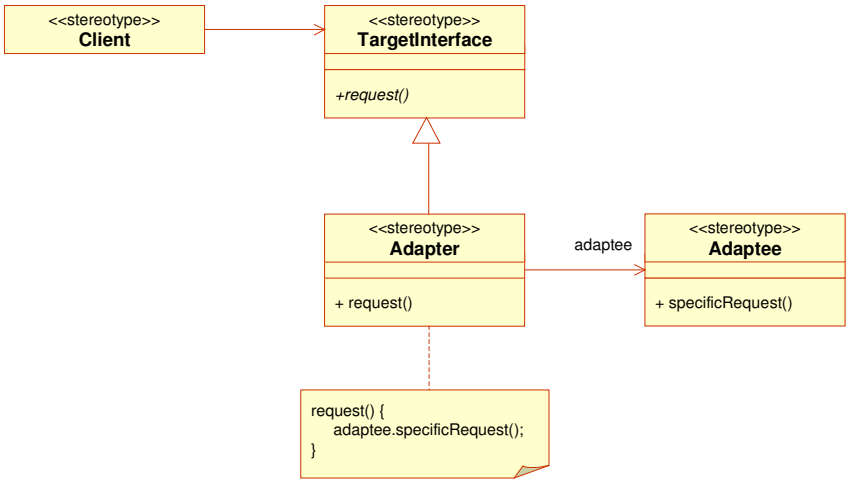
- Object Adapter: si crea una nuova classe (RectangleObjectAdapter) che implementa l'interfaccia richiesta, e che possiede al suo interno un'istanza della classe a riutilizzare. Le operazioni della nuova classe fanno invocazioni ai metodi dell'oggetto interno.

6.4. Struttura del Pattern

Per il *Class Adapter*:

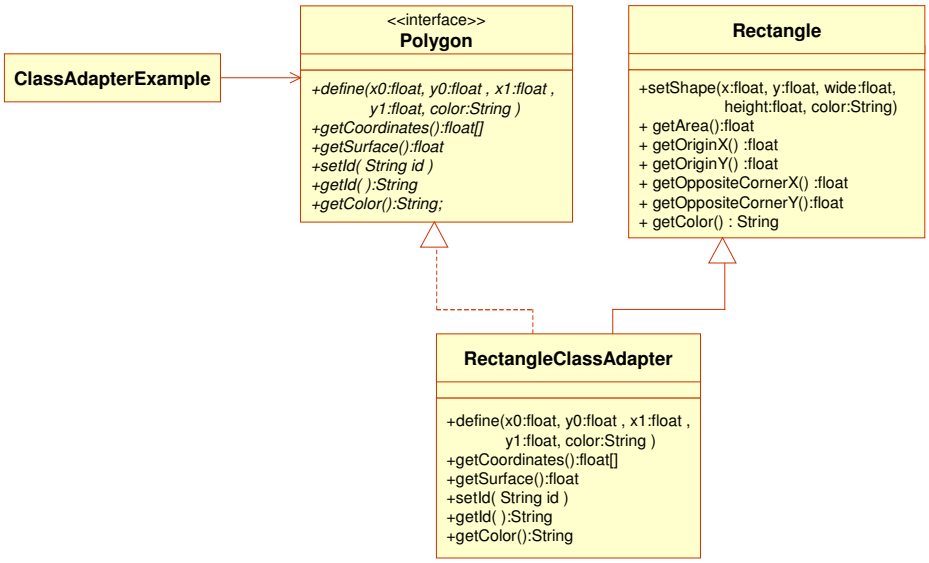


Per l'*Object Adapter*:

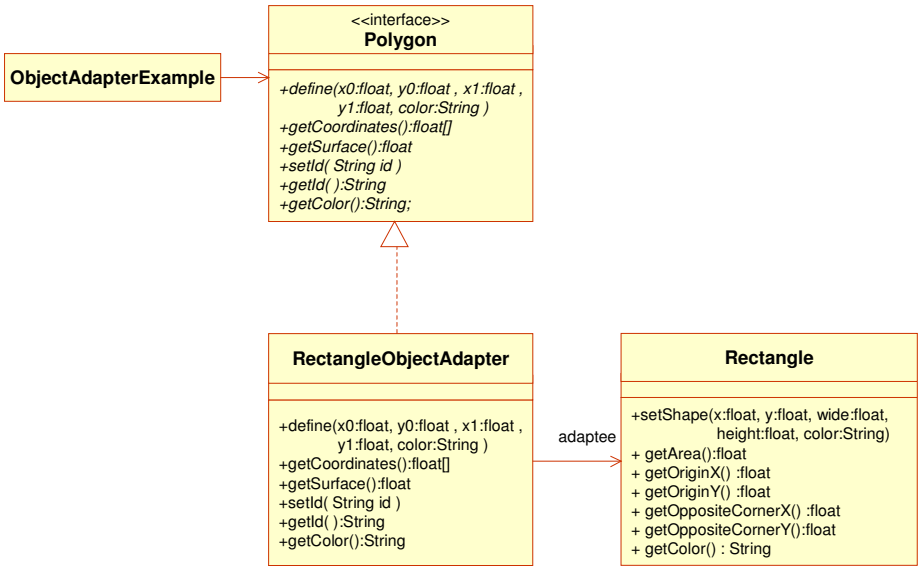


Schema del modello

Nel caso del Class Adapter il modello da implementare è il seguente:



Invece, se viene utilizzato l'Object Adapter, il modello corrisponde a:



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**: classe Rectangle.
  - Implementa una interfaccia che deve essere adattata.
- **Adapter**: classi RectangleClassAdapter e RectangleObjectAdapter.
  - Adatta l'interfaccia dell'**Adaptee** verso la **TargetInterface**.

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.

a) 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 it's " +
                             block.getColor() );

    }
}
```

b) 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.getOppositeCornerX();
       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`).

Esecuzione dell'esempio

```
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
```

6.5. 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`.

7. Bridge

(GoF pag. 151)

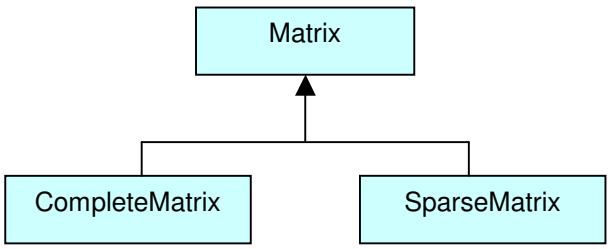
7.1. Descrizione

Separa un'astrazione dalla sua implementazione, in modo che entrambe possano variare indipendentemente.

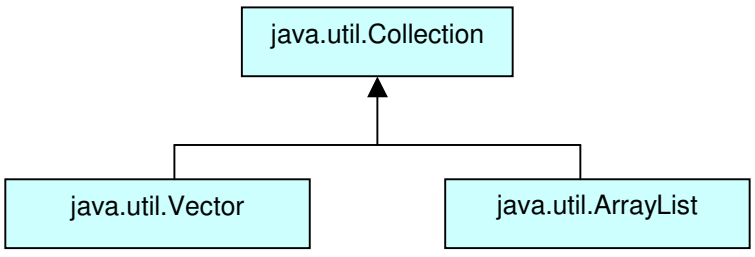
7.2. Esempio

Nello sviluppo di una libreria di classi per supporto allo sviluppo di applicazioni matematiche, si ha definito la possibilità di gestire due tipologie di matrici numeriche, come raffinamento dell'astrazione Matrix, che fornisce le operazioni generali di gestione di dati matriciali:

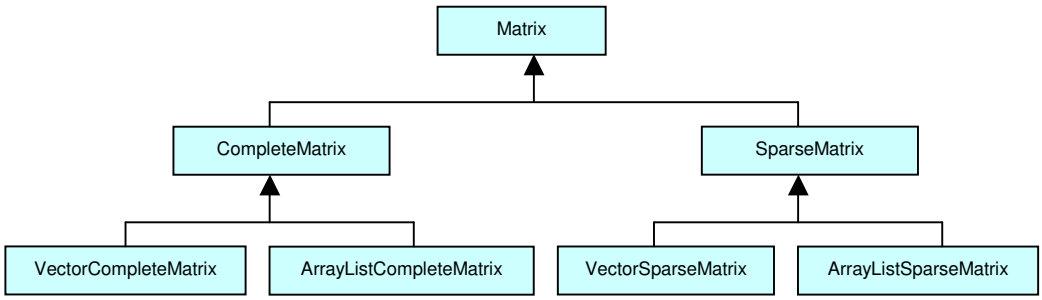
- La matrice completa, nella quale tutte le celle sono rappresentate come oggetti all'interno di essa (CompleteMatrix).
- La matrice *sparse,* che soltanto ha nei suoi interni oggetti contenenti i valori diversi di zero (SparseMatrix).



Indipendentemente del tipo dei raffinamento di matrice a utilizzare, le celle della matrice (che saranno rappresentate da oggetti aventi come attributi le proprie coordinate e il valore da tenere) dovranno essere immagazzinati in qualche tipologia di collezione d'oggetti. Si consideri che gli sviluppatori della libreria considerano due tipi di collezioni per implementare il supporto delle celle, tra le diverse tipologie di collezioni fornite da Java: `java.util.Vector` e `java.util.ArrayList`.



Se si vuole tenere le due tipologie di matrici insieme alle due implementazioni di collezione, in modo di poter scegliere la configurazione più adatta ad ogni singola situazione, una progettazione semplicista delle classi condurrebbe alla seguente gerarchia:



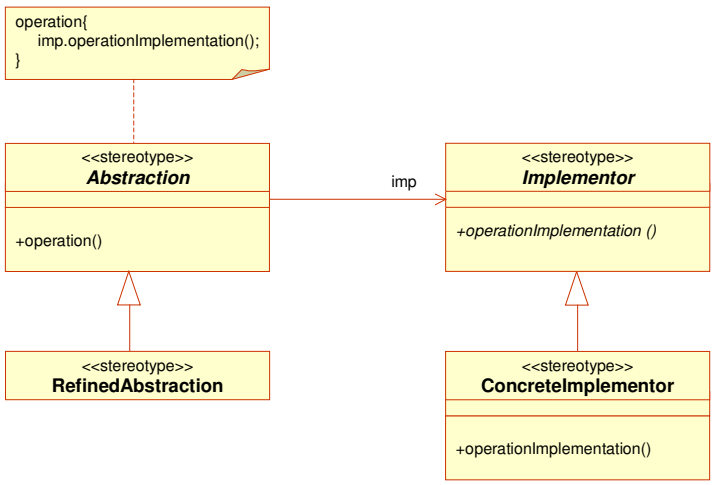
Sebbene intuitiva, questa gerarchia non è efficiente perché ogni volta che è necessario aggiungere un nuovo tipo di matrice, bisogna creare due classi specifiche, una per ogni tipo d'implementazione (`ArrayList` o `Vector`). Da un'altra parte, se si decide utilizzare un nuova implementazione di collezione, sarà necessario aggiungere una nuova classe per ogni tipo di matrice esistente.

Il problema consiste nella definizione di un modo di rappresentare e gestire efficientemente entrambe gerarchie, in modo tale che una possa variare senza avere impatto sull'altra.

7.3. Descrizione della soluzione offerta dal pattern

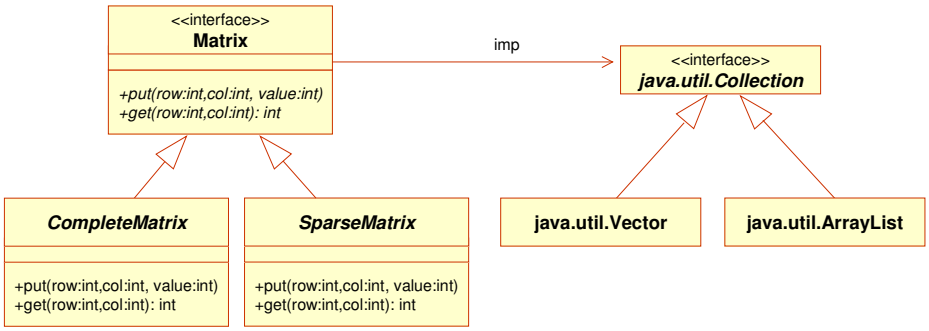
Il "*Bridge*" pattern suggerisce la separazione dell'astrazione (gerarchia di Matrix) dall'implementazione (gerarchia di Collection), in gerarchie diverse, legando oggetti della seconda a quelli della prima, tramite un relazione di conmposizione. In questo modo ogni oggetto della gerarchia di Matrix sarà configurato con un particolare oggetto Collection da utilizzare.

7.4. Struttura del Pattern



7.5. Applicazione del pattern

Schema del modello



Partecipanti

- **Abstraction**: interfaccia Matrix.
  - Specifica l'interfaccia dell'astrazione.
  - Gestisce un riferimento ad un oggetto **Implementor**.
- **RefinedAbstraction**: classi CompleteMatrix e SparseMatrix.
  - Implementano l'interfaccia definita dall'**Abstraction**.
- **Implementor**: Interfaccia Collection.
  - Specifica l'interfaccia definita per le classi di implementazione.
- **ConcreteImplementor**: classi Vector e ArrayList.
  - Implementano l'interfaccia **Implementor**.

Descrizione del codice

La classe astratta `Matrix` (**Abstraction**) fornisce i metodi di base comuni alla gestione della collezione di supporto per gli oggetti corrispondenti alle celle della matrice (in questo caso sono stati dichiarati `protected`)[8]. Definisce, anche, l'interfaccia che dovranno implementare le **RefinedAbstraction** (metodi pubblici):

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

public abstract class Matrix {

   int rows, cols;
   Collection data ;

   protected Matrix( int rows, int cols, Collection collection ) {
      this.rows = rows;
      this.cols = cols;
      data = collection;
   }
```

[8] Per l'uso dell'iterator presente nel codice di alcuni metodi, si veda il Java Tutorial [15], oppure il pattern "Iterator" in questa relazione.

```
   protected MatrixCell createPosition( int row, int col )
                                 throws MatrixIndexOutOfBoundsException {
      MatrixCell mc = getPosition( row, col );
      if( mc == null )
         mc = new MatrixCell( row, col );
      data.add( mc );
      return mc;

   }

   protected void deletePosition( MatrixCell toDelete )
                                 throws MatrixIndexOutOfBoundsException {
      data.remove( toDelete );
   }

   protected MatrixCell getPosition( int row, int col )
                                 throws MatrixIndexOutOfBoundsException {
      if( row < 0 || row >= this.rows || col < 0 || col >= this.cols )
        throw new MatrixIndexOutOfBoundsException();

      Iterator it = data.iterator();
      while( it.hasNext() ) {
          MatrixCell mc = (MatrixCell) it.next();
          if( mc.row == row && mc.col == col )
            return mc;
      }
      return null;
   }

   public abstract void put( int row, int col, int value )
                                  throws MatrixIndexOutOfBoundsException;
   public abstract int get( int row, int col )
                                  throws MatrixIndexOutOfBoundsException;

}
```

La classe `CompleteMatrix` è una **RefinedAbstraction** che rappresenta la matrice in forma completa (cioè quella che istanza oggetti per tutte le celle). In questo caso il costruttore accetta come parametro un riferimento ad un oggetto corrispondente ad un tipo particolare di collezione, che però viene trattato all'interno di questa classe tramite l'interfaccia `java.util.Collection`:

```
import java.util.Collection;

public class CompleteMatrix extends Matrix {

   public CompleteMatrix( int rows, int cols, Collection collection ) {
      super( rows, cols, collection );
      for(int i = 0 ; i< rows; i++ )
         for(int j = 0 ; j< cols; j++ )
           createPosition( i, j );

   }

   public void put( int row, int col, int value )
                                  throws MatrixIndexOutOfBoundsException{
      MatrixCell cell = getPosition( row, col );
      cell.value = value;
   }

   public int get( int row, int col )
                                 throws MatrixIndexOutOfBoundsException {
      MatrixCell cell = getPosition( row, col );
      return cell.value;
   }

}
```

La matrice sparse è un'altra **RefinedAbstraction,** implementata nella classe `SparseMatrix`:

```
import java.util.Collection;

public class SparseMatrix extends Matrix {

   public SparseMatrix( int rows, int cols, Collection collection ) {
        super( rows, cols, collection );
   }

   public void put( int row, int col, int value )
                                throws MatrixIndexOutOfBoundsException {
      MatrixCell cell = getPosition( row, col );
      if( cell != null )
         if( value == 0 ) {
             deletePosition( cell );
         } else {
             cell.value = value;
         }
      else
         if( value != 0 ) {
           cell = createPosition( row, col );
           cell.value = value;
         }
   }

   public int get( int row, int col )
                                throws MatrixIndexOutOfBoundsException {
      MatrixCell cell = getPosition( row, col );
      if( cell == null)
         return 0;
      else
         return cell.value;
   }

}
```

Si presenta adesso la classe `MatrixCell` che rappresenta la cella:

```
public class MatrixCell {

   public int row, col, value;

   public MatrixCell( int r, int c ) {
      row = r;
      col = c;
      value = 0;
   }

}
```

E la classe `MatrixIndexOutOfBoundsException`, che non fa altro che caratterizzare gli errori di accesso a posizioni fuori i limiti della matrice:

```
public class MatrixIndexOutOfBoundsException extends RuntimeException {}
```

Finalmente, il codice che esegue l'esempio. Si noti come è questo che crea le istanze dei particolari tipi di collezioni che saranno utilizzate per dare supporto ad ogni tipo di Matrice.

```
import java.util.Vector;
import java.util.ArrayList;

public class BridgeExample {

  public static final int ROWS = 3;
  public static final int COLS = 4;

 public static void main( String[] arg ) {
   CompleteMatrix matrixCV = new
```

```
                              CompleteMatrix( ROWS, COLS, new Vector() );
    System.out.println( "Complete Matrix with Vector:");
    matrixCV.put( 1, 2, 1 );
    matrixCV.put( 2, 1, 2 );
    matrixCV.put( 0, 3, 3 );
    matrixCV.put( 1, 2, 0 );
    for(int i = 0 ; i< ROWS; i++ ) {
       for(int j = 0 ; j< COLS; j++ )
            System.out.print( matrixCV.get( i, j )+" " );
       System.out.println();
    }

    SparseMatrix matrixSV = new SparseMatrix( ROWS, COLS, new Vector() );
    System.out.println( "Sparse Matrix with Vector:");
    matrixSV.put( 1, 2, 1 );
    matrixSV.put( 2, 1, 2 );
    matrixSV.put( 0, 3, 3 );
    matrixSV.put( 1, 2, 0 );
    for(int i = 0 ; i< ROWS; i++ ) {
       for(int j = 0 ; j< COLS; j++ )
            System.out.print( matrixSV.get( i, j )+" " );
       System.out.println();
    }

    CompleteMatrix matrixCA = new
                           CompleteMatrix( ROWS, COLS, new ArrayList() );
    System.out.println( "Complete Matrix with ArrayList:");
    matrixCA.put( 1, 2, 1 );
    matrixCA.put( 2, 1, 2 );
    matrixCA.put( 0, 3, 3 );
    matrixCA.put( 1, 2, 0 );
    for(int i = 0 ; i< ROWS; i++ ) {
       for(int j = 0 ; j< COLS; j++ )
            System.out.print( matrixCA.get( i, j )+" " );
       System.out.println();
    }

    SparseMatrix matrixSA = new
                            SparseMatrix(  ROWS, COLS, new ArrayList() );
    System.out.println( "Sparse Matrix with ArrayList:");
    matrixSA.put( 1, 2, 1 );
    matrixSA.put( 2, 1, 2 );
    matrixSA.put( 0, 3, 3 );
    matrixSA.put( 1, 2, 0 );
    for(int i = 0 ; i< ROWS; i++ ) {
       for(int j = 0 ; j< COLS; j++ )
            System.out.print( matrixSA.get( i, j )+" " );
       System.out.println();
    }
  }
}
```

Osservazioni sull'esempio

E' discutibile la scelta delle strutture dati utilizzate nell'implementazione di questa matrice. Non è stato l'obiettivo esemplificare una implementazione ottimale.

Si noti che in questo esempio, la classe con il ruolo di **Abstraction**, oltre a specificare l'interfaccia per le **RefinedAbstraction**, fornisce le operazioni di base per agire sull'**Implementor**.

Esecuzione dell'esempio

```
C:\Design Patterns\Structural\Bridge>java BridgeExample

Complete Matrix with Vector:
0 0 0 3
0 0 0 0
0 2 0 0

Sparse Matrix with Vector:
0 0 0 3
0 0 0 0
0 2 0 0

Complete Matrix with ArrayList:
0 0 0 3
0 0 0 0
0 2 0 0

Sparse Matrix with ArrayList:
0 0 0 3
0 0 0 0
0 2 0 0
```

7.6. Osservazioni sull'implementazione in Java

Non ci sono aspetti particolari da segnare.

8. Composite

(Gof pag. 163)

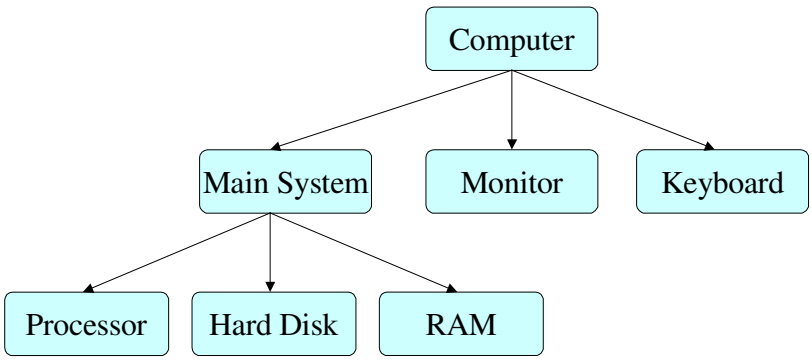
8.1. Descrizione

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.

8.2. 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").

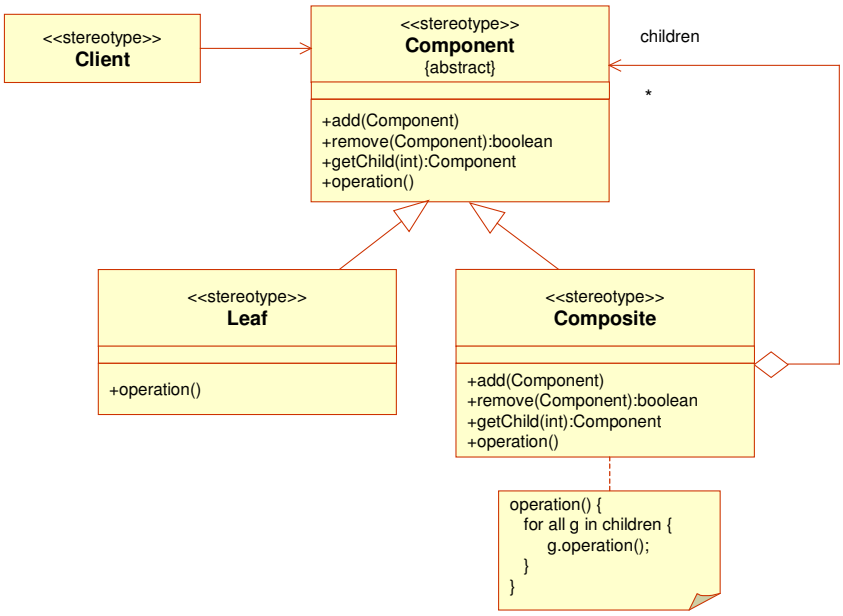


Il problema è la rappresentazione omogenea di tutti gli elementi presenti del magazzino, sia dei singoli componenti, sia di quelli composti da altri componenti.

8.3. Descrizione della soluzione offerta dal pattern

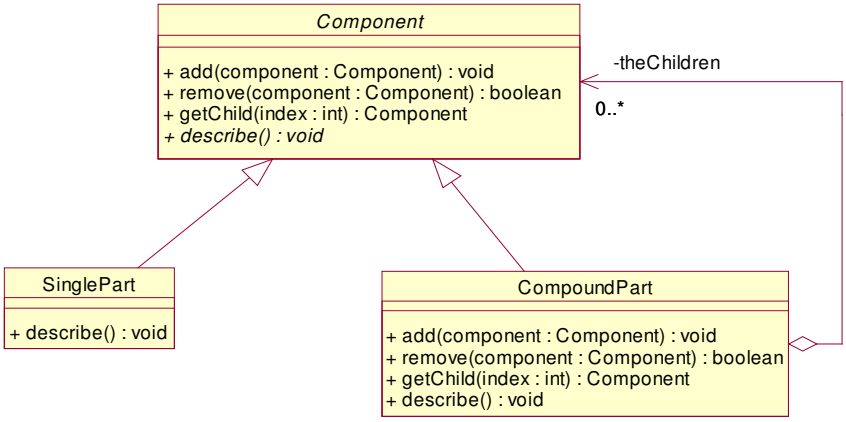
Il pattern “Composite” definisce la classe astratta componente (Component) che deve essere estesa in due sottoclassi: una che rappresenta i singoli componenti (Leaf), e un’altra (Composite) che rappresenta i componenti composti, e che si implementa come contenitore di componenti. Il fatto che quest’ultima 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).

8.4. Struttura del Pattern



8.5. Applicazione del pattern

Schema del modello



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)[9]. 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();
```

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

```
  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( );
  }

  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" );
  }
}
```

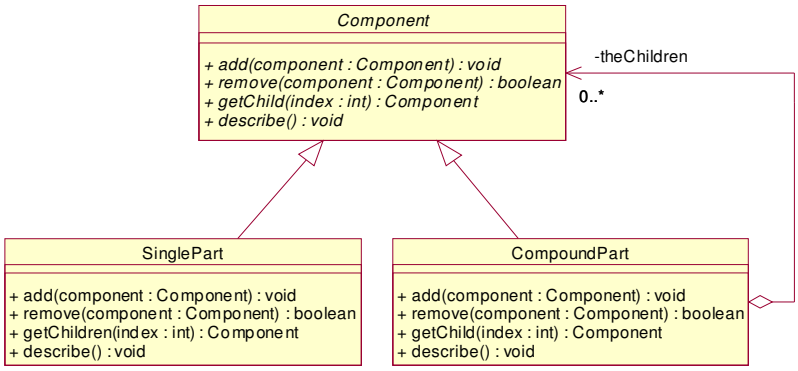
L'applicazione `CompositeExample` fa le veci del **Client** che gestisce i diversi tipi di pezzi, tramite l'interfaccia comune fornita dalla classe `Component`. Nella prima parte dell'esecuzione si creano dei pezzi singoli (`monitor`, `keyboard`, `processor`, `ram` e `hardDisk`), dopodiché viene creato 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 );
      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 seguente figura:



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 );
  }
}
```

Esecuzione dell'esempio

```
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)
```

8.6. Osservazioni sull'implementazione in Java

Non ci sono aspetti particolari da tenere in conto.

9. Decorator

(GoF pag. 117)

9.1. Descrizione

Aggiunge dinamicamente responsabilità addizionali ad un oggetto. In questo modo si possono estendere le funzionalità d'oggetti particolari senza coinvolgere complete classi.

9.2. Esempio

Si pensi ad un modello di oggetti che rappresenta gli impiegati (Employee) di una azienda. Tra gli impiegati, ad esempio, esistono gli Ingegneri (Engineer) che implementano le operazioni definite per gli impiegati, secondo le proprie caratteristiche.

Il sistema comprende la possibilità di investire gli impiegati con delle responsabilità aggiuntive, ad esempio, quando un impiegato diventa capoufficio (Administrative Manager), oppure, quando viene assegnato alla direzione di un progetto (Project Manager), essendo entrambe responsabilità non esculenti tra di loro.

Questi cambiamenti di tipologia di alcuni impiegati coinvolgono modifiche delle responsabilità definite per gli oggetti, alterandone le esistenti o aggiungendone nuove. Per questa ragione, sarebbe di interesse definire un modo per aggiungere dinamicamente nuove responsabilità ad oggetto specifico, eventualmente con la ulteriore possibilità di toglierle.

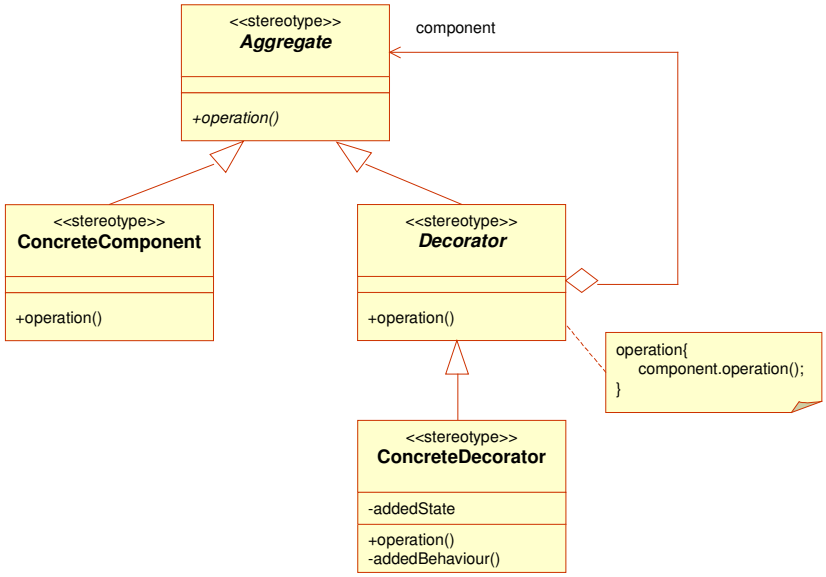
9.3. Descrizione della soluzione offerta dal pattern

Il pattern suggerisce la creazione di *wrapper classes* (Decorator) che racchiudono gli oggetti ai quali si vuole aggiungere le nuove responsabilità. Questi ultimi oggetti, insieme ai Decorator devono implementare una interfaccia comune, in modo che l'applicazione possa continuare ad interagire con gli oggetti decorati.

Per una stessa interfaccia possono esserci più Decorator, ad esempio, per investire i ruoli di capoufficio e di responsabile di un progetto.

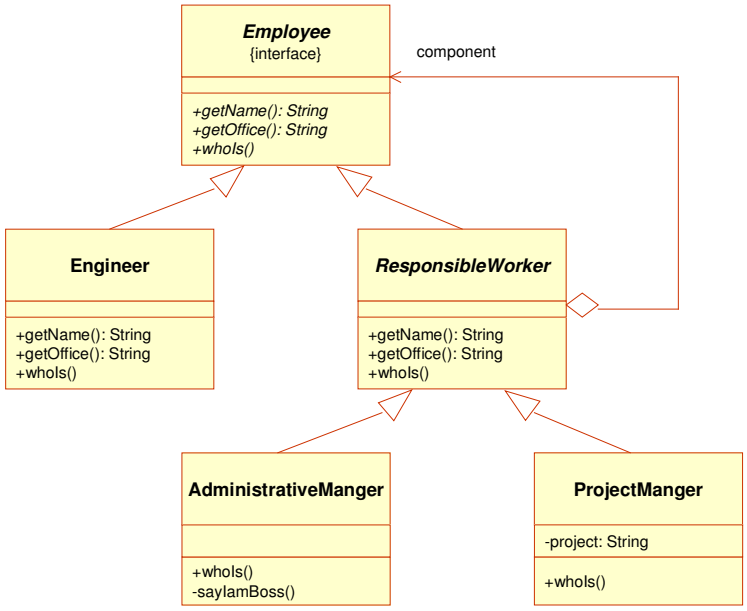
Il fatto che Decorator e oggetti decorati implementino la stessa interfaccia, consente anche l'applicazione di un Decorator ad un altro oggetto già decorato, ottenendo in questo modo la sovrapposizione di funzioni (ad esempio, un impiegato potrebbe essere investito come capoufficio e responsabile di un progetto contemporaneamente).

9.4. Struttura del Pattern



9.5. Applicazione del pattern

Schema del modello



Partecipanti

- **Component**: classe Employee.
  - Specifica l'interfaccia degli oggetti che possono avere delle responsabilità aggiunte dinamicamente.
- **ConcreteComponent**: classe Engineer.

  - Implementa l'oggetto in cui si possono aggiungere nuove responsabilità.

- **Decorator**: classe ResponsibleWorker.
  - Possiede un riferimento all'oggetto Component e specifica una interfaccia concordante con l'interfaccia Component.
- **ConcreteDecorator**: classe AdministrativeManager e ProjectManager.
  - Aggiunge nuove responsabilità al Component.

Descrizione del codice

Tutti i componenti di questo modello implementano l'interfaccia `Employee`, che specifica soltanto tre metodi: uno per restituire il proprio nome (`getName`), un altro per restituire il nome dell'ufficio al quale appartiene (`getOffice`), e un ultimo che fa la presentazione di se stesso (`whoIs`).

```
public interface Employee {

  public String getName();
  public String getOffice();
  public void whoIs();

}
```

La classe `Engineer` implementa l'interfaccia `Employee`:

```
public class Engineer implements Employee {

   private String name, office;

   public Engineer( String nam, String off ) {
      name    = nam;
      office  = off;
   }

   public String getName() {
      return name ;
   }

   public String getOffice() {
      return office ;
   }

   public void whoIs() {
      System.out.println( "I am " + getName() + ", and I am with the "
                          + getOffice() +".");
   };

}
```

La classe astratta `ResponsibleWorker` corrisponde al **Decorator** del modello. Contiene il codice necessario per immagazzinare al suo interno l'oggetto decorato (**Component**), e mappa verso di lui le operazioni richieste. Si noti che questa classe implementa l'intrerfaccia `Employee`, e al suo interno utilizza questa stessa interfaccia per comunicare con il **Component**.

```
abstract class ResponsibleWorker implements Employee {

    protected Employee responsible;
```

```
    public ResponsibleWorker(Employee employee) {
        responsible = employee;
    }
    public String getName() {
      return responsible.getName();
    }

    public String getOffice() {
      return responsible.getOffice();
    }

    public void whoIs() {
      responsible.whoIs();
    }

}
```

La responsabilità particolare che riguarda le funzioni di un capoufficio sono codificate nell'`AdministrativeManager`. Questa classe estende le funzioni del **Decorator**, particolarmente aggiungendo l'operazione `sayIamBoss`, che viene chiamata come parte della ridefinizione del metodo `whoIs` :

```
public class AdministrativeManager extends ResponsibleWorker {

   public AdministrativeManager( Employee empl ) {
      super( empl );
   }

   public void whoIs() {
       sayIamBoss();
       super.whoIs();
   }

   private void sayIamBoss(){
       System.out.print( "I am a boss. " );
   }

}
```

La classe `ProjectManager`, invece estende le variabili di stato dell'oggetto decorato, e modifica il comportamento dell'oggetto (metodo `whoIs`):

```
public class ProjectManager extends ResponsibleWorker {

   private String project;

   public ProjectManager( Employee empl, String proj ) {
     super( empl );
     project = proj;
   }

   public void whoIs() {
     super.whoIs();
     System.out.println( "I am the Manager of the Project:" + project );
   }

}
```

Finalmente si presenta l'applicazione che crea un `Employee`, lo investe come `AdministrativeManager` (per indicare che l'impiegato è capoufficio), e poi, due volte come `ProjectManager` (per segnare che l'impiegato è anche responsabile della gestione di due progetti):

```
public class DecoratorExample1 {
```

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

     Employee thisWillBeFamous = new Engineer( "William Gateway",
                                               "Programming Department" );
     System.out.println( "Who are you?");
     thisWillBeFamous.whoIs();

     thisWillBeFamous = new AdministrativeManager( thisWillBeFamous );
     System.out.println( "Who are you now?");
     thisWillBeFamous.whoIs();

     thisWillBeFamous = new ProjectManager( thisWillBeFamous,
                                      "D.O.S.- Doors Operating System" );
     System.out.println( "Who are you now?");
     thisWillBeFamous.whoIs();

     thisWillBeFamous = new ProjectManager( thisWillBeFamous,
                                    "EveryoneLoggedToInternet Explorer" );
     System.out.println( "Who are you now?");
     thisWillBeFamous.whoIs();

  }

}
```

Osservazioni sull'esempio

Si noti che in questo esempio furono creati **ConcreteDecorators** che:

- Estendono operazioni esistenti del **Component** (metodo `sayIamBoss` del `AdministrativeManager`).
- Estendono lo stato del Component (attributo `project` del `ProjectManager`).

Deve notarsi che prima un **ConcreteDecorator** fu applicato su un **ConcreteComponent** (`AdministrativeManager` su `Engineer`). Poi su quel **ConcreteDecorator** fu applicato un altro **ConcreteDecorator** (`ProjectManager`). Finalmente, su quest'ultimo venne applicato un'altra volta un'altra istanza dello stesso **ConcreteDecorator**.

Esecuzione dell'esempio

```
C:\Design Patterns\Structural\Decorator\Exmple1>java DecoratorExample1

Who are you?
I am William Gateway, and I am with the Programming Department.

Who are you now?
I am a boss. I am William Gateway, and I am with the Programming
Department.

Who are you now?
I am a boss. I am William Gateway, and I am with the Programming
Department.
I am the Manager of the Project: D.O.S.- Doors Operating System

Who are you now?
I am a boss. I am William Gateway, and I am with the Programming
Department.
I am the Manager of the Project: D.O.S.- Doors Operating System
I am the Manager of the Project: EveryoneLoggedToInternet Explorer
```

Un altro Esempio

Un uso che può avere un **Decorator** è sincronizzare un oggetto costruito originalmente per essere utilizzato in un ambiente single-threading, quando si vuol portare a un ambiente d'esecuzione multi-threading.

Per esemplificare questo uso si ha definito la interfaccia `DiagonalDraggablePoint` (**Component**) che fornisce la firma dei metodi per muovere un punto diagonalmente una distanza specificata, e per scrivere la posizione attuale.

```
public interface DiagonalDraggablePoint {

     public void moveDiagonal( int distance, String draggerName );
     public void currentPosition( );

}
```

La classe `SequentialPoint` (**ConcreteComponent**) implementa la citata interfaccia. Si noti che il metodo `moveDiagonal` soltanto aggiunge lo stesso valore alle variabili d'istanza `x` e `y`, e stampa questi valori prima di modificarli. Questo significa che sempre `x` e `y` dovrebbero essere numericamente uguali.

```
public class SequentialPoint implements DiagonalDraggablePoint {

     private int x, y;

     public SequentialPoint( ) {
         this.x = 0;
         this.y = 0;
     }

     public void moveDiagonal( int distance, String draggerName ) {
         int aux = x + distance ;
         System.out.println( "Moved by " + draggerName +
                             " - Origin x=" + x + " y=" + y );
         x = aux;
         y = y + distance;
     }


     public void currentPosition( ) {
         System.out.println( "Current position : x=" + x + " y=" + y );
     }


}
```

In una esecuzione multi-threaded il codice anteriore ha il gravissimo problema che non blocca l'oggetto nel processo di movimento (`moveDiagonal`), così che se il thread in esecuzione è interrotto en questa fase, alcune delle variabili potrebbero essere aggiornate da un altro thread, prima che il controllo ritornassi, ottenendosi come risultato valori diversi di `x` e `y`. Il problema si può risolvere creando un **Decorator** (`SynchronizedPoint`) che implementi l'interfaccia `DiagonalDraggablePoint` e che sincronizzi il codice rischioso:

```
public class SynchronizedPoint implements DiagonalDraggablePoint {

     DiagonalDraggablePoint theSequentialPoint;

     public SynchronizedPoint(DiagonalDraggablePoint np) {
```

```
        theSequentialPoint = np;
     }


     public void moveDiagonal( int distance, String draggerName ) {
         synchronized(theSequentialPoint) {
         theSequentialPoint.moveDiagonal( distance, draggerName );
         }
      }

     public void currentPosition( ) {
        theSequentialPoint.currentPosition();
     }

}
```

L'applicazione `DecoratorExample2` fa una prima prova d'uso con l'oggetto sequenziale che è contemporaneamente mosso da due thread. In una seconda fase si fa la stessa prova, ma adesso con l'oggetto sincronizzato:

```
public class DecoratorExample2 {

  public static void main( String[] arg ) {

     System.out.println( "Non synchronized point:" );
     DiagonalDraggablePoint p = new SequentialPoint();
     PointDragger mp1 = new PointDragger( p, "Thread 1" );
     PointDragger mp2 = new PointDragger( p, "Thread 2" );
     Thread t1 = new Thread( mp1 );
     Thread t2 = new Thread( mp2 );
     t1.start();
     t2.start();
     while( t1.isAlive() || t2.isAlive()  );
     p.currentPosition();

     System.out.println( "Synchronized point:" );
     p = new SynchronizedPoint( new SequentialPoint() );
     mp1 = new PointDragger( p, "Thread 1" );
     mp2 = new PointDragger( p, "Thread 2" );
     t1 = new Thread( mp1 );
     t2 = new Thread( mp2 );
     t1.start();
     t2.start();
     while( t1.isAlive() || t2.isAlive()  );
     p.currentPosition();

  }

}

class PointDragger implements Runnable {
  DiagonalDraggablePoint point;
  String name ;
  public PointDragger( DiagonalDraggablePoint p, String nom ) {
     point = p;
     name  = nom;
  }
  public void run() {
    for( int i=1; i < 5; i++ ) {
       point.moveDiagonal( 1, name );
    }
  }
}
```

Esecuzione dell'esempio

L'esecuzione del programma mostra che l'oggetto non protetto dal meccanismo di blocco, raggiunge uno stato inconsistente, che in questo caso si è raggiunto quando il secondo thread, nel suo secondo movimento mostra che i valori di `x` e `y` sono diversi. L'utilizzo, invece, del oggetto protetto consente di osservare che questo mai raggiunge uno stato inconsistente.

```
C:\Design Patterns\Structural\Decorator\Exmpl2>java DecoratorExample2

Non synchronized point:
Moved by Thread 1 - Origin x=0 y=0
Moved by Thread 1 - Origin x=1 y=1
Moved by Thread 2 - Origin x=1 y=1
Moved by Thread 1 - Origin x=2 y=2
Moved by Thread 2 - Origin x=2 y=3
Moved by Thread 1 - Origin x=3 y=4
Moved by Thread 2 - Origin x=3 y=5
Moved by Thread 2 - Origin x=4 y=7
Current position : x=5 y=8

Synchronized point:
Moved by Thread 1 - Origin x=0 y=0
Moved by Thread 1 - Origin x=1 y=1
Moved by Thread 2 - Origin x=2 y=2
Moved by Thread 1 - Origin x=3 y=3
Moved by Thread 2 - Origin x=4 y=4
Moved by Thread 1 - Origin x=5 y=5
Moved by Thread 2 - Origin x=6 y=6
Moved by Thread 2 - Origin x=7 y=7
Current position : x=8 y=8
```

9.6. Osservazioni sull'implementazione in Java

Java utilizza Decorator simili al presentato in questo ultimo esempio per la sincronizzazione delle collezioni.

10. Facade

(GoF pag. 185)

10.1. Descrizione

Fornisce una interfaccia unificata per un insieme di interfacce di un sottosistema, rendendo più facile l'uso di quest'ultimo.

10.2. Esempio

Un applicativo Java abilitato ad accettare dati forniti tramite l'interfaccia di consola, deve gestire l'interazione di un insieme di classi di Input/Output e di conversione di tipi di dati. Per esempio, se si vuole leggere un numero si richiedono le seguenti istruzioni:

```
//Crea una istanza di un oggetto lettore
BufferedReader br = new BufferedReader( new
                                        InputStreamReader( System.in ) );

String aString ="";

// La lettura può generare una eccezione
try {
   aString = br.readLine();
} catch( IOException e ) { }
```

Il numero così letto è, in realtà, una stringa. Se dopo si vuole utilizzare come dato numerico, ad esempio, dentro un oggetto `Integer`, la stringa letta dovrà interagire con questa classe:

```
// Si converte il valore letto a int
Integer anInteger = new Integer( aString );
```

Queste operazioni sono abbastanza comuni nelle applicazioni. Si deve notare, però, che ogni volta che si vuole inserire dati di tipo numerico il programmatore deve interagire con le seguenti classi:

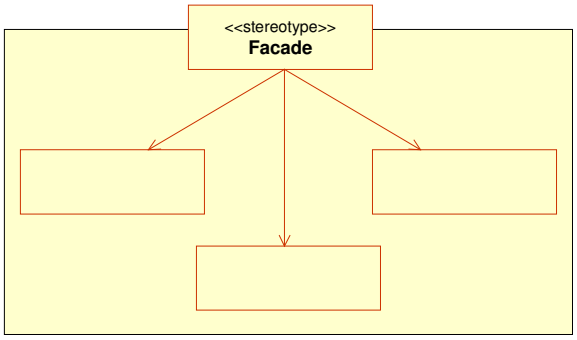
- BufferedReader
- InputStreamReader
- System
- IOException
- String, Integer, Double, Float, ecc.

Sarebbe auspicabile una forma più facile di interagire con loro.

10.3. Descrizione della soluzione offerta dal pattern

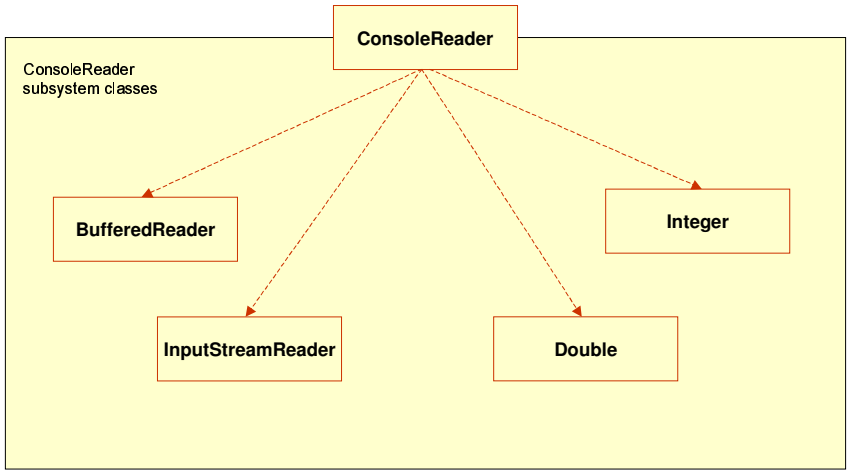
Il "*Facade*" pattern suggerisce la creazione di un oggetto che presentia un'interfaccia semplificata al cliente, ma in grado di gestire tutta la complessità delle interazioni tra gli oggetti delle diverse classi per compiere l'obbiettivo desiderato.

10.4. Struttura del pattern



10.5. Applicazione del pattern

Schema del modello



Partecipanti

- **Facade**: classe ConsoleReader.
  - Ha conoscenza delle funzionalità di ogni classe del sottosistema.
  - Delega agli appropriati oggetti del sottosistema ogni richiesta pervenuta dall'esterno.
- **Subsystem classes**: classi BufferedReader, InputStreamReader, Doble e Integer, tra altre.
  - Implementano le funzionalità del sottosistema
  - Gestiscono le attività assegnate dal **Facade**.
  - Non hanno riferimenti verso il **Facade**.

Descrizione del codice

Si implementa la classe `ConsoleReader`, che offre i metodi per l'inserimento dati tramite consola. I metodi restituiscono oggetti `String`, `Integer` o `Double` (tutti e tre appartenenti al pakage `java.lang`), secondo la necessità del chiamante.

Nel caso che il dato letto non sia di un formato valido, ad esempio, se l'utente fornisce una lettera quando si aspetta come input un valore intero, si ha deciso di fornire due diversi tipi di comportamento: lanciare un'eccezione `ConsoleReaderException` (modo sicuro) o generare l'oggetto con valore nullo, vale dire, zero (modo no sicuro). L'utente dovrà decidere quale strategia adoperare, al momento d'istanziare il `ConsolerReader`, secondo le proprie esigenze.

L'utilizzo di questo oggetto di lettura a interfaccia di lettura sarà ad esempio:

```
//La variabile statica SECURE_MODE_OFF indica che l'oggetto lettore
//trascurerà gli errori.
ConsoleReader rdr = new ConsoleReader( ConsoleReader.SECURE_MODE_OFF );

// Legge una stringa
String text = reader.readString();

// Legge un numero e restituisce un oggetto di tipo Integer:
Integer numberInt = reader.readInteger();

// Legge un numero e restituisce un oggetto di tipo Double:
Double numberDouble = reader.readDouble();
```

Il codice del `ConsoleReader`, è il seguente:

```
import java.io.*;
public class ConsoleReader {

  public static final boolean SECURE_MODE_ON  = true;
  public static final boolean SECURE_MODE_OFF = false;

  BufferedReader br;
  boolean secureMode;

  public ConsoleReader() {
      this( SECURE_MODE_ON );
  }

  public ConsoleReader(boolean secMod ) {
      br = new BufferedReader( new InputStreamReader( System.in ) );
      secureMode = secMod;
  }

  public String readString() {
      try {
         return br.readLine();
      } catch (Exception e) {
         if ( secureMode )
            throw new ConsoleReaderException( e );
         return "";
      }
  }

  public Integer readInteger() {
      Integer theInteger = null ;
      try {
         theInteger = new Integer( br.readLine() );
      } catch (Exception e) {
         if ( secureMode )
```

```
            throw new ConsoleReaderException( e );
         theInteger = new Integer( 0 );
        }
        return theInteger;
    }

    public Double readDouble() {
        Double theDouble = null ;
        try {
           theDouble = new Double( br.readLine() );
        } catch (Exception e) {
           if ( secureMode )
              throw new ConsoleReaderException( e );
         theDouble = new Double( 0 );
        }
        return theDouble;
    }

}
```

La classe `ConsoleReaderException` estende `RuntimeException`, e si presenta di seguito:

```
public class ConsoleReaderException extends RuntimeException {
    public ConsoleReaderException( Exception e ) {
        super( e.toString() );
    }
}
```

Adesso ripresenta il codice del programma di prova, che nella prima parte lavora col lettore in modo insicuro (errori non noti), e poi con lo stesso in modo sicuro. Si osservi quanto si sia semplificato l'inserimento di dati, in relazione con i tradizionali meccanismi di input:

```
public class FacadeExample {

  public static void main( String[] arg ) {

        ConsoleReader reader = new ConsoleReader(
                                        ConsoleReader.SECURE_MODE_OFF );

        System.out.println( "This is the 'Insecure mode': " +
                            "the system can't detect invalid inputs." );
        System.out.println( "Enter a string : ");
        String text = reader.readString();
        System.out.println( "You wrote: " + text );

        System.out.println( "Enter an Integer : ");
        Integer oInteger = reader.readInteger();
        System.out.println( "You wrote: " + oInteger );

        System.out.println( "Enter a Double : ");
        Double oDouble = reader.readDouble();
        System.out.println( "You wrote: " + oDouble );

        System.out.println( "This is the 'Secure mode': " +
                            "the system raises an exception when it " +
                            "detects invalid inputs." );

        reader = new ConsoleReader( );
        System.out.println( "Enter a string : ");
        text = reader.readString();
        System.out.println( "You wrote: " + text );

        System.out.println( "Enter an Integer : ");
        oInteger = reader.readInteger();
        System.out.println( "You wrote: " + oInteger );

        System.out.println( "Enter a Double : ");
        oDouble = reader.readDouble();
        System.out.println( "You wrote: " + oDouble );

  }
}
```

Osservazioni sull'esempio

La gestione delle eccezioni si presenta solo per completezza della progettazione della classe.

Si possono aggiungere metodi per restituire tipi primitivi (`int`, `double`, ecc.) e altri oggetti (`Byte`, `Float`, ecc.)

Esecuzione dell'esempio

Si trascrive un dialogo possibile nell'esecuzione del codice dell'esempio. Tanto nel modo insicuro come nel sicuro, si fornisce un dato non valido.

```
C:\ Design Patterns\Structural\Facade>java FacadeExample

This is the 'Insecure mode': the system can't detect invalid inputs.

Enter a string :
this is a test
You wrote: this is a test

Enter an Integer :
x
You wrote: 0

Enter a Double :
3
You wrote: 3.0

This is the 'Secure mode': the system raises an exception when it
detects invalid inputs.

Enter a string :
another test
You wrote: another test

Enter an Integer :
3
You wrote: 3

Enter a Double :
x
Exception in thread "main" ConsoleReaderException:
java.lang.NumberFormatException: x
        at ConsoleReader.readDouble(ConsoleReader.java:47)
        at FacadeExample.main(FacadeExample.java:32)
```

10.6. Osservazioni sull'implementazione in Java

Non ci sono considerazioni particolari nell'implementazione Java.

11. Flyweight

(GoF pag. 195)

11.1. Descrizione

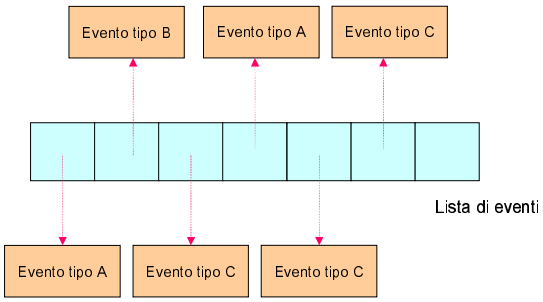
Definisce un meccanismo per compartire oggetti, in modo di fare un uso più efficiente delle risorse, particolarmente, l'utilizzo di memoria.

11.2. Esempio

Un sistema di controllo deve gestire in tempo reale una coda di eventi (Event) che permanentemente vengono generati. Gli eventi, rappresentati come oggetti, devono essere analizzati da un controllore, che deve reagire opportunamente. Ad esempio, un modo nel quale il controllore potrebbe gestirli sarebbe:

```
SomeEventType eventKind1, eventKind2, eventKind3,…;
…

// Creates objects for each eventKind
…

// Receives an event
SomeEventType receivedEvent = eventListener.listen();

// Classifies the event and do something
if( receivedEvent.equals( eventKind1 ) )
   // action 1
else if( receivedEvent.equals( eventKind2 ) )
   // action 2
else if( receivedEvent.equals( eventKind3 ) )
   // action 3
else…
   // etc…
…
```

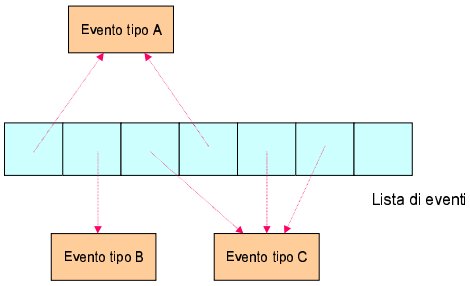
La gestione degli eventi appena descritta, crea nella memoria diversi oggetti identici che poi vengono confrontati tra di loro per determinare la uguaglianza:



In sistemi come quello descritto, normalmente la tipologia degli eventi è limitata a un insieme relativamente ridotto il cui stato in parte non varia da una istanza all'altra.Date queste caratteristiche, questo approccio comporta inefficienze dal punto di vista della gestione della memoria (importante da tenere in conto nel caso dei dispositivi di capacità limitata), e anche nelle comparazioni di tipo di eventi, se si confronta con l'approccio fornito dal "*Flyweight*" pattern.

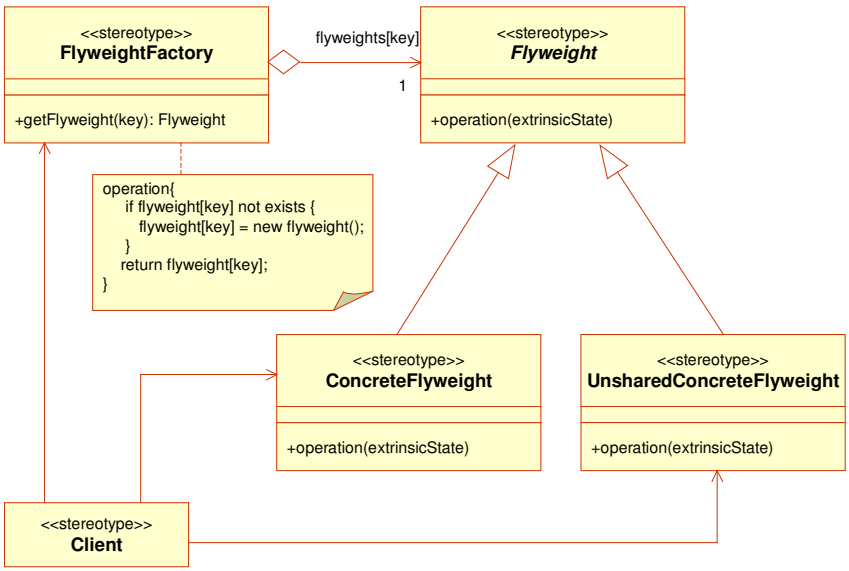
11.3. Descrizione della soluzione offerta dal pattern

Questo pattern suggerisce la creazione dei "*flyweight*" che sono degli oggetti condivisi che possono essere utilizzati in diversi contesti contemporaneamente, dovuto al fatto che in questi contesti condividono lo stato interno, e quello che non è interno si può esternalizzare. In altre parole, questo pattern affronta la creazione di un gruppo di singoli oggetti diversi per rappresentare le tipologie di eventi, i cui riferimenti vengono riutilizzati per rappresentare ogni istanza di evento presente nel sistema:



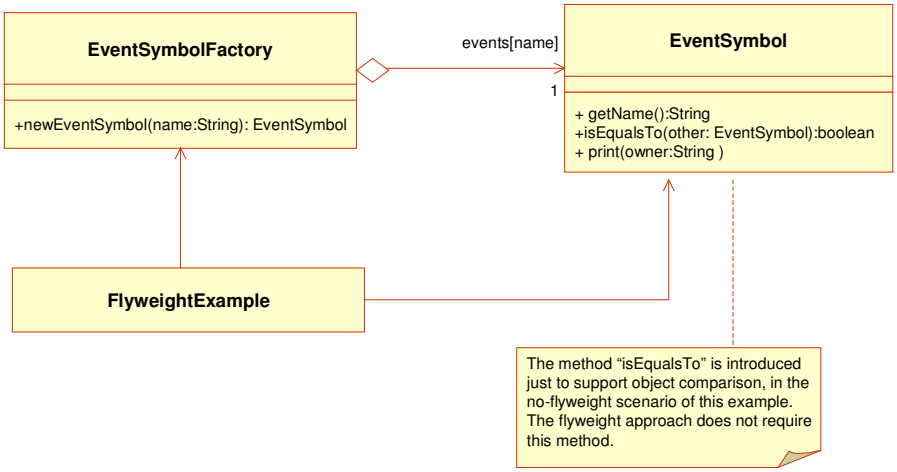
Questo approccio apporta due vantaggi all'efficienza del sistema presentato nell'esempio: riduce l'utilizzo della memoria (importante da tenere in conto nel caso dei dispositivi di capacità limitata), e rende più veloci le comparazioni di tipo, poiché vengono confrontati i riferimenti, non gli oggetti stessi.

11.4. Struttura del Pattern



11.5. Applicazione del pattern

Schema del modello



Partecipanti

- **Flyweight**: non necessario in questo esempio.
  - Specifica l'interfaccia tramite la quale i flyweight ricevono e agiscono sullo stato estrinseco.
- **ConcreteFlyweight**: classe EventSymbol.
  - Implementa l'interfaccia dei flyweight e immagazzina lo stato interno del flyweight. Lo stato interno deve essere intrinseco, vale dire, indipendente del contesto del **ConcreteFlyweight**.
- **UnsharedConcreteFlyweight**: non presente in questo esempio.
  - Rappresentano i tipi di flyweight non condivisi.
- **FlyweightFactory**: classe EventSymbolFactory.
  - Crea e gestisce gli oggetti flyweight.
  - Assicura la condivisione dei flyweight. Quando un cliente richiede un flyweight, la FlyweightFactory fornisce un riferimento a una istanza esistente o ne crea una, se non esiste nessuna.
- **Client**: applicativo FlyweightExample.
  - Possiede riferimenti verso i flyweight.
  - Calcola o immagazzina lo stato estrinseco dei flyweight.

Descrizione del codice

Il codice descritto di seguito confronta nello stesso esempio l'approccio di gestione di oggetti non condivisi (in altre parole, l'approccio "normale"), con quello suggerito dal Flyweight pattern.

Si presenta per primo la classe `EventSymbolFactory` (**FlyweightFactory**) incaricata di fornire i riferimenti ai flyweight, ogni volta che un oggetto di questo tipo viene richiesto. Per fare ciò la `EventSymbolFactory` fornisce il metodo `newEventSymbol` che accetta come parametro una stringa che caratterizza il nome dell'evento richiesto. Al suo interno, questa classe possiede una hashtable dove vengono inseriti tutti gli eventi creati. Ricevuta una richiesta di restituzione di un evento, viene scandita la hashtable per trovare un riferimento al corrispondente evento se creato in precedenza, oppure viene istanziato un nuovo oggetto evento e addizionato a questa collezione.

```
import java.util.Hashtable;
public class EventSymbolFactory {

   private static Hashtable events = new Hashtable();

   public static EventSymbol newEventSymbol( String evnt ) {

       EventSymbol evntRef = (EventSymbol) events.get( evnt );
       if( evntRef == null ) {
         evntRef = new EventSymbol( evnt );
         events.put( evnt, evntRef );
       }
       return evntRef;
   }

}
```

I `ConcreteFlyweight` sono gli eventi rappresentati da oggetti della classe `EventSymbol`. L'attributo `name` serve a tenere lo stato interno condiviso tra le istanze che rappresentano lo stesso evento, intanto che il metodo `print` agisce sullo stato esterno, ricevuto come parametro dal contesto di chiamata. Il metodo `isEqualsTo` è fornito soltanto per determinare l'uguaglianza tra due oggetti `EventSymbol` nello scenario senza flyweight (vale dire, con istanze non condivise) proposto in questo esempio, e non viene utilizzato quando l'evento è gestito come flyweight. Si noti che in questo esempio non è stata specificata una interfaccia separata per i **Flyweight**, vale dire l'`EventSymbol` (**ConcreteFlyweight**) dichiara e implementa l'interfaccia.

```
public class EventSymbol {

    private String name;

    public EventSymbol( String name ) {
        this.name = name;
    }

    public String getName() {
        return name;
    }


    public void print( String owner ) {
        System.out.println("Event: " + name + " Owner: "+ owner );
    }

    public boolean isEqualsTo( EventSymbol otherEvent ) {
        return name.equals( otherEvent.getName() );
    }

}
```

Si presenta per primo l'esempio `FlyweightComparisonExample`, che fa un *benchmark* della velocità di creazione e comparazione di 1.000.000 di oggetti della classe `EventSymbol`, che rappresentano tre tipi di eventi diversi. Si confronta la gestione non condivisa oggetti, con quella condivisa.

```
import java.util.Date;
public class FlyweightComparisonExample {

    public static void main (String[] arg) {

      EventSymbol[] eventList = new EventSymbol[1000000];
      Date tStart, tFinish;
      long prepTime, compTime;

      // No-flyweight usage statistics
      System.out.println( "Testing creation and comparison of "
                          + "unshared EventSymbol instances" );
      tStart = new Date();
      for(int i=0 ; i < eventList.length; i++) {
         double r = Math.random();
         if(r < .3)
             eventList[i] = new EventSymbol( "EVENT 1" );
         else if( r < .6 )
             eventList[i] = new EventSymbol( "EVENT 2" );
         else
             eventList[i] = new EventSymbol( "EVENT 3" );
      }
      tFinish = new Date();
      prepTime = tFinish.getTime() - tStart.getTime();

      int evn1count=0, evn2count=0, evn3count=0;

      EventSymbol compEvent1 = new EventSymbol( "EVENT 1" );
      EventSymbol compEvent2 = new EventSymbol( "EVENT 2" );
      EventSymbol compEvent3 = new EventSymbol( "EVENT 3" );
      tStart = new Date();
      for(int i=0 ; i < eventList.length; i++)
         if( eventList[i].isEqualsTo( compEvent1 ) )
             evn1count++;
         else if( eventList[i].isEqualsTo( compEvent2 ) )
             evn2count++;
         else
             evn3count++;
      tFinish = new Date() ;
      compTime = tFinish.getTime() - tStart.getTime();

      System.out.println( "E1: " + evn1count +"  E2: " + evn2count +
                          "  E3: " + evn3count );
      System.out.println( "Filling   time: " + prepTime );
      System.out.println( "Computing time: " + compTime );
      System.out.println( "Total time    : " + (prepTime + compTime) );

      // Flyweight usage statistics
      System.out.println( "Testing creation and comparison of shared"+
                          "EventSymbol instances" );
      tStart = new Date();
      for(int i=0 ; i < eventList.length; i++) {
         double r = Math.random();
         if(r < .3)
            eventList[i] = EventSymbolFactory.newEventSymbol("EVENT 1");
         else if( r < .6 )
            eventList[i] = EventSymbolFactory.newEventSymbol("EVENT 2");
          else
            eventList[i] = EventSymbolFactory.newEventSymbol("EVENT 3");
      }
      tFinish = new Date();

      prepTime = tFinish.getTime() - tStart.getTime();
      evn1count = 0; evn2count = 0; evn3count = 0;

      tStart = new Date();
      compEvent1 = EventSymbolFactory.newEventSymbol( "EVENT 1" );
      compEvent2 = EventSymbolFactory.newEventSymbol( "EVENT 2" );
```

```
      compEvent3 = EventSymbolFactory.newEventSymbol( "EVENT 3" );
      for(int i=0 ; i < eventList.length; i++)
         if( eventList[i] == compEvent1  )
             evn1count++;
         else if( eventList[i] == compEvent2 )
             evn2count++;
         else
             evn3count++;
      tFinish = new Date();

      compTime = tFinish.getTime() - tStart.getTime();
      System.out.println( "E1: " + evn1count +"  E2: " + evn2count +
                                                "  E3: " + evn3count );
      System.out.println( "Filling   time: " + prepTime );
      System.out.println( "Computing time: " + compTime );
      System.out.println( "Total time    : " + (prepTime + compTime) );

   }

}
```

Si offre un secondo esempio, `FlyweightExternalStateExample`, che mette in evidenza l'uso dello stato esterno dell'oggetto, che in questo caso è il nome del generatore dell'evento. Questo nome è gestito dall'applicativo, che lo fornisce come parametro al metodo `print` del `EventSymbol`.

```
import java.util.Date;
public class FlyweightExternalStateExample {

  public static void main (String[] arg) {

    System.out.println( "Exemplifies the external state usage." );
    EventSymbol evnt;
    for(int i=0 ; i < 10; i++) {
      String owner =  Math.random()< .5 ? "Generator A" : "Generator B";
      double r = Math.random();
      if(r < .3)
           evnt = EventSymbolFactory.newEventSymbol( "EVENT 1" );
      else if( r < .6 )
           evnt = EventSymbolFactory.newEventSymbol( "EVENT 2" );
      else
           evnt = EventSymbolFactory.newEventSymbol( "EVENT 3" );
      evnt.print( owner );
    }

  }

}
```

Osservazioni sull'esempio

In questo esempio non è stato utilizzato un **UnsharedConcreteFlyweight**, e perciò non si è voluta l'interfaccia separata del **Flyweight**. Il **ConcreteFlyweight** specifica e implementa la propria interfaccia.

Questo esempio aggiunge la velocità di gestione, come un altro motivo di utilizzo del Flyweight pattern, all'elenco segnato da Gamma et Al., il quale solo tiene conto dei risparmi di memoria.

L'utilizzo di riferimenti condivisi per la gestione di oggetti simili è spesso utilizzata nella progettazione di sistemi di controllo che devono agire in *real-time*, dato che la velocità di confronto risulta notevolmente ridotta.

E' da testare più accuratamente l'efficienza del processo di creazione di oggetti, dato che la diversità di oggetti presenti nell'esempio è molto ridotta, e perciò le ricerche all'interno della hashtable della factory sono

abbastanza brevi. Sarebbe di interesse testare il comportamento con un insieme più grande di oggetti.

Esecuzione dell'esempio

Si offre per primo il risultato dell'esecuzione del programma `FlyweightComparisonExample`. I tempi sono presentati in millisecondi.

```
C:\Design Patterns\Structural\Flyweight>java
FlyweightComparisonExample

Testing creation and comparison of unshared EventSymbol instances
E1: 299393  E2: 300166  E3: 400441
Filling   time: 1392
Computing time: 231
Total time    : 1623

Testing creation and comparison of shared EventSymbol instances
E1: 300703  E2: 299501  E3: 399796
Filling   time: 440
Computing time: 30
Total time    : 470
```

Si presenta adesso l'output prodotto dal programma `FlyweightExternalStateExample`.

```
C:\Design Patterns\Structural\Flyweight>java
FlyweightExternalStateExample

Exemplifies the external state usage.
Event: EVENT 2 Owner: Generator B
Event: EVENT 3 Owner: Generator B
Event: EVENT 3 Owner: Generator A
Event: EVENT 2 Owner: Generator A
Event: EVENT 3 Owner: Generator A
Event: EVENT 3 Owner: Generator B
Event: EVENT 2 Owner: Generator A
Event: EVENT 3 Owner: Generator A
Event: EVENT 3 Owner: Generator A
Event: EVENT 2 Owner: Generator B
```

11.6. Osservazioni sull'implementazione in Java

Non ci sono considerazioni particolare riguardanti l'implementazione di questo pattern in Java.

12. Proxy

(GoF pag. 207)

12.1. Descrizione

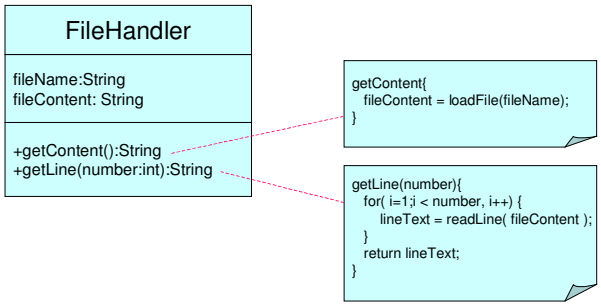
Fornisce una rappresentazione di un oggetto di accesso difficile o che richiede un tempo importante per l'accesso o creazione. Il Proxy consente di posticipare l'accesso o creazione al momento in cui sia davvero richiesto

12.2. 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.
-



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.

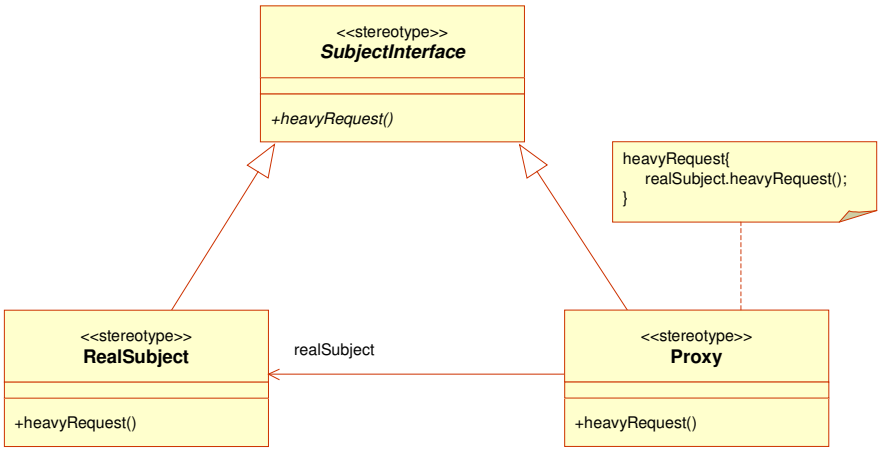
E' di interesse poter gestire più efficientemente questa classe, senza modificare la sua implementazione.

12.3. 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*".

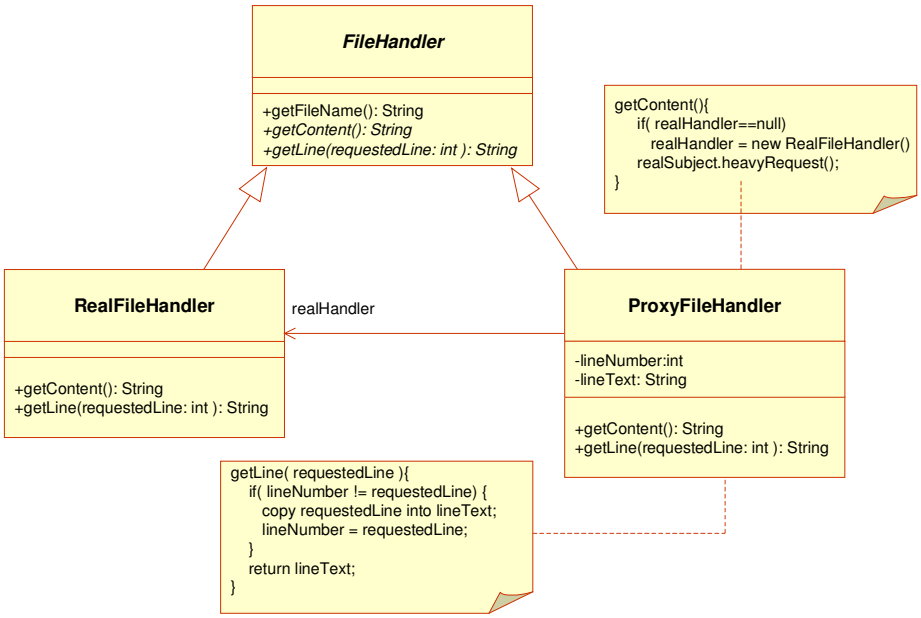
Come altra funzionalità, a questo proxy potrebbe essere aggiunta la possibilità di imagazinare temporaneamente l'ultima riga restituita dal metodo "getLine", in modo che due richieste successive della stesa linea comportino solo una ricerca, riducendo lo spreco di tempo di elaborazione. Il proxy che offre questa funzionalità viene chiamato "*caché proxy*".

12.4. Struttura del Pattern



12.5. Applicazione del pattern

Schema del modello



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.
- **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 ) );
    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`.

```
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 "
```

12.6. 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".

13. Chain of Responsibility

(GoF pag. 223)

13.1. 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.

13.2. Esempio

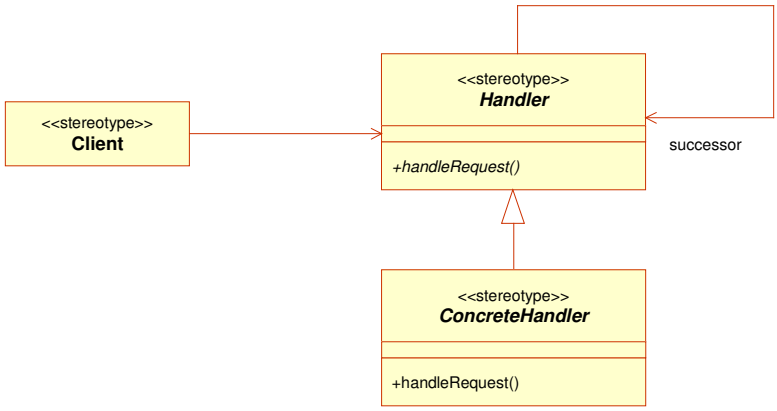
Una azienda commerciale deve gestire le richieste di credito dei clienti (*customers*). Internamente l'azienda si organizza in diversi livelli di responsabilità. Al livello più basso (*vendor*) viene consentito l'approvazione di richieste fino a un importo determinato. Le richieste che superano questo importo vanno gestite da un livello superiore (*sales manager*), 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.

13.3. Descrizione della soluzione offerta dal pattern

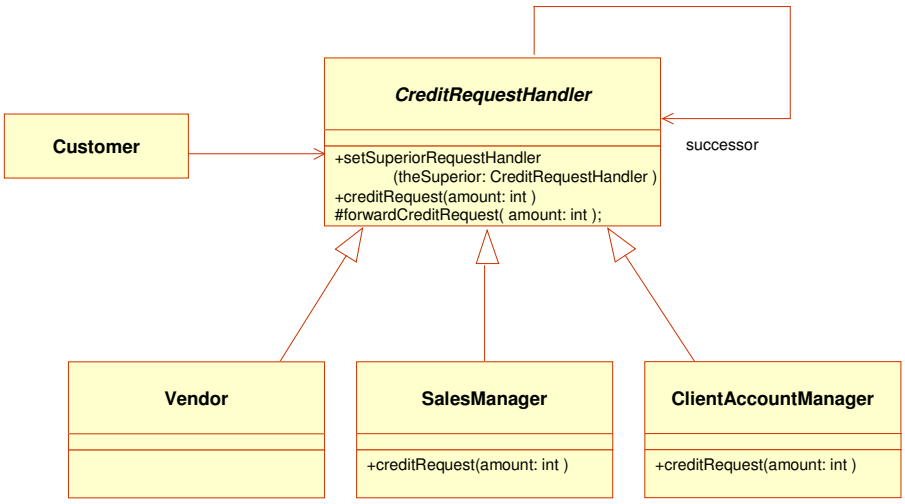
Il "Chain of responsibility" pattern propone la costruzione di una catena di oggetti responsabili della gestione delle richieste pervenute dai clienti. Quando un oggetto della catena riceve una richiesta, analizza se corrisponde a lui gestirla, o, altrimenti, inoltrarla al seguente oggetto dentro la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

13.4. Struttura del pattern



13.5. Applicazione del pattern

Schema del modello



Partecipanti

- **Handler**: classe CreditRequestHandler.
  - Specifica una interfaccia per la gestione delle richieste.
  - In modo opzionale, implementa un riferimento a un oggetto successore.
- **ConcreteHandler**: classi Vendor, SalesManager e ClientAccountManager.
  - 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();
      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.

Esecuzione dell'esempio

```
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)
```

13.6. 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.

14. Command

(GoF pag. 233)

14.1. Descrizione

Incapsula una richiesta all'interno di un oggetto, consentendo la parametrizzazione dei clienti con diverse tipologie di richieste, accodare o rintracciare le richieste, oppure supportare operazioni di *undo*.

14.2. Esempio

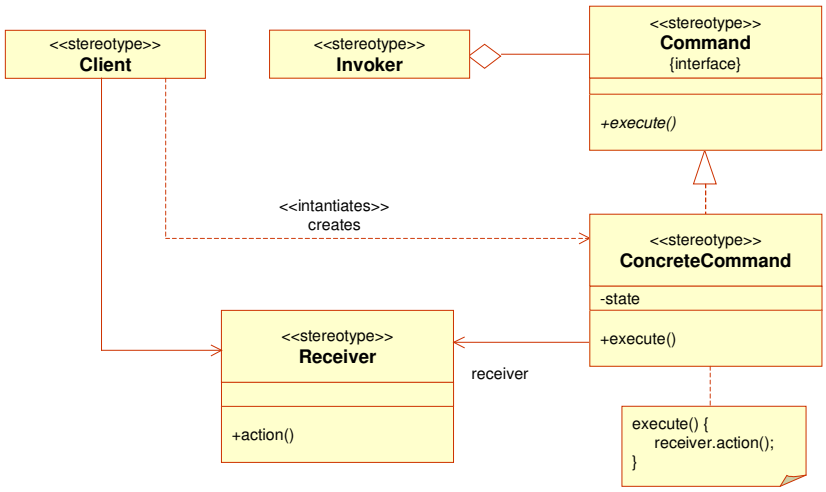
Un telecomando universale consente di gestire diversi tipi di elettrodomestici (TV di diversi modelli e marche, Hi-Fi, ecc.). Durante la fase di progettazione del telecomando si riesce a definire il numero di tasti e la etichetta di ogni singolo tasto, ma non le operazione concrete da eseguire (vale dire, il segnale particolare da spedire all'elettrodomestico). Si spera che queste operazioni vengano aggiunte, ulteriormente, al momento di configurare il telecomando (ad esempio, l'utente potrebbe scaricare da internet le classi che li servono per i propri elettrodomestici).

Il problema consiste nella progettazione di una classe (telecomando), in grado di inoltrare richieste verso oggetti che saranno noti solo in fasi successive di sviluppo.

14.3. Descrizione della soluzione offerta dal pattern

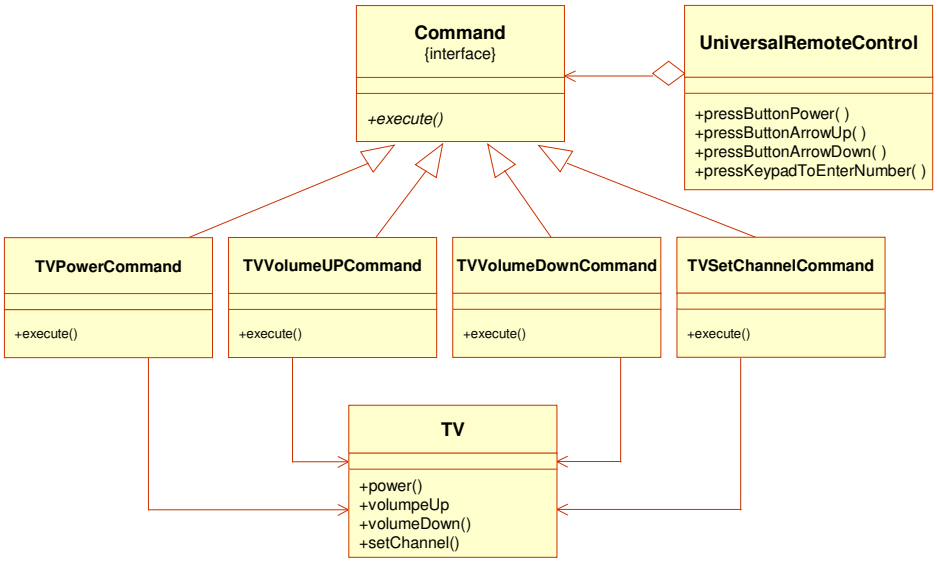
Il pattern "*Command*" cambia le richieste di operazioni su oggetti non specificati, trasformando le richieste stesse in oggetti. La base di questo pattern è la classe astratta Command che specifica l'interfaccia per l'esecuzione delle operazioni, contenente un metodo astratto "execute". Gli specifici comandi che verranno eseguiti estendono questa classe e specificano la coppia destinatario/azione, immagazzinando il destinatario come variabile di istanza e implementando il metodo "execute" per invocare la richiesta.

14.4. Struttura del Pattern



14.5. Applicazione del modello

Schema del modello



Partecipanti

- **Command**: classe Command.
  - Dichiara una interfaccia per l'esecuzione di una operazione.
- **ConcreteCommand**: classi TVPowerCommand, TVVolumeUpCommand, TVVolumeDownCommand e TVSetChannelCommand.
  - Ognuno definisce una associazione tra un **Receiver** e una azione.
  - Implementa l'operazione Execute, invocando la corrispondente operazione del **Receiver**.
- **Invoker**: classe UniversalRemoteControl.
  - Richiede tramite il **Command** l'esecuzione di una operazione.
- **Receiver**: classe TV.
  - Porta avanti le operazioni richieste. Qualunque classe può essere **Receiver**.

Descrizione del codice

La classe `TV` è il **Receiver** delle operazioni richieste dall'`UniversalRemoteControl`, che possono essere accensione o spegnimento, aumento o diminuzione del volume, o cambio del canale. Come **Receiver**, questa classe non richiede l'implementazione di nessuna interfaccia particolare:

```
public class TV {

  public static final int ON  = 1;
  public static final int OFF = 0;
```

```
  private int power   = OFF;
  private int volume  = 0;
  private int channel = 2;


  public void power() {
      if( power == OFF ) {
         power = ON;
         System.out.println( "The TV is ON." );
      } else {
         power = OFF;
         System.out.println( "The TV is OFF." );
      }
  }

  public void volumeUp() {
      if( power == ON) {
         if( volume < 10 )
           volume++;
         System.out.println( "Current volume level is " + volume );
      } else
         System.out.println( "You must turn on the TV.");
  }

  public void volumeDown() {
      if( power == ON ) {
         if( volume > 0 )
           volume--;
         System.out.println( "Current volume level is " + volume );
      } else0
         System.out.println( "You must turn on the TV.");
  }

  public void setChannel( int ch ) {
      if( power == ON ) {
        if( ch >= 2 && ch < 140 )
          channel = ch;
        System.out.println( "Current channel is " + channel );
      } else
        System.out.println( "You must turn on the TV.");
  }

}
```

La classe astratta `Command` specifica l'interfaccia con la quale comunica lo `UniversalRemoteControl` (**Invoker**) le sue richieste alla TV (**Receiver**), e che ogni singolo **ConcreteCommand** deve implementare. Si noti che questa interfaccia non fornisce riferimento alcuno verso qualche particolare **Receiver**:

```
public interface Command {
        public abstract void execute ( );
}
```

L'`UniversalRemoteControl` viene configurato nel costruttore con i diversi **ConcreteCommand** incaricati di inoltrare le richieste alla `TV`. Si noti che le operazioni richiamano il metodo `execute` di ogni comando:

```
public class UniversalRemoteControl {

   private Command buttonPower,
                   buttonArrowUp,
                   buttonArrowDown,
                   keypadToEnterNumber,
                   buttonScan;

   public UniversalRemoteControl( Command pw, Command au,
                                  Command ad, Command nu ) {
          buttonPower          = pw;
          buttonArrowUp        = au;
          buttonArrowDown      = ad;
          keypadToEnterNumber  = nu;
```

```
        }

   public void pressButtonPower( ) {
        buttonPower.execute ( ) ;
   }

   public void pressButtonArrowUp( ) {
        buttonArrowUp.execute ( ) ;
   }

   public void pressButtonArrowDown( ) {
        buttonArrowDown.execute ( ) ;
   }

   public void pressKeypadToEnterNumber( ) {

        keypadToEnterNumber.execute ( ) ;
   }

   public void pressBack() {

        keypadToEnterNumber.undo() ;

   }
}
```

L'implementazione dei **ConcreteCommand** si presenta di seguito. Si noti che ognuno di essi implementa nel metodo `execute` l'invocazione a una particolare operazione sull'oggetto TV.

```
class TVPowerCommand implements Command {

    private TV theTV;

    public TVPowerCommand ( TV someTV ) {
        theTV = someTV ;
    }

    public void execute() {
        theTV.power();
    }

}
```

```
class TVVolumeUpCommand implements Command {

    private TV theTV;

    public TVVolumeUpCommand ( TV someTV ) {
        theTV =  someTV ;
    }

    public void execute() {
        theTV.volumeUp();
    }

}
```

```
class TVVolumeDownCommand implements Command {

    private TV theTV;

    public TVVolumeDownCommand ( TV someTV ) {
        theTV =  someTV ;
    }

    public void execute() {
        theTV.volumeDown();
    }

}
```

```
import java.io.*;
class TVSetChannelCommand implements Command {

   private TV theTV;

    public TVSetChannelCommand ( TV someTV ) {
        theTV = someTV;
    }

    public void execute() {
        int currentChannel = acceptNumber();
        theTV.setChannel( currentChannel );
    }

    private int acceptNumber() {
      int chInput = 0;
      BufferedReader br = new BufferedReader(
                              new InputStreamReader( System.in ) );
      try {
      System.out.println( "Enter the channel number:" );
        chInput =  Integer.parseInt( br.readLine() );
      } catch( IOException e ) {
        System.out.println( "Error" );
      }
      return chInput;
    }

}
```

Finalmente si offre il codice dell'applicativo che crea un oggetto della classe `TV`, gestito da un `UniversalRemoteControl`, precedentemente configurato con dei concreti comandi:

```
import java.io.*;
public class CommandExample {

   public static void main(String[] args) throws IOException {

      TV aCommonTV = new TV();
      Command  tvpower = new TVPowerCommand( aCommonTV );
      Command  tvVolUp = new TVVolumeUpCommand( aCommonTV );
      Command  tvVolDn = new TVVolumeDownCommand( aCommonTV );
      Command  tvSetCh = new TVSetChannelCommand( aCommonTV );
      UniversalRemoteControl remote =
                                  new UniversalRemoteControl( tvpower,
                                                              tvVolUp,
                                                              tvVolDn,
                                                               tvSetCh );

      BufferedReader br = new BufferedReader( new
                                       InputStreamReader( System.in ) );
      int nOption = 0;
      while( nOption != 9 ) {
        System.out.println( "Select a button to press or 9 to exit" );
        System.out.println( "1.- POWER" );
        System.out.println( "2.- ARROW UP" );
        System.out.println( "3.- ARROW DOWN" );
        System.out.println( "4.- NUMERIC KEYPAD" );
        nOption = Integer.parseInt( br.readLine() );
        switch (nOption) {
          case 1 : remote.pressButtonPower();
                   break;
          case 2 : remote.pressButtonArrowUp();
                   break;
          case 3 : remote.pressButtonArrowDown();
                   break;
          case 4 : remote.pressKeypadToEnterNumber();
                   break;
        }
      }
  }

}
```

Osservazioni sull'esempio

In questo esempio non è stata implementata l'operazione di *undo*, che questo pattern consente di implementare.

Da un'altra parte, si noti che l'interfaccia **Command** non ha riferimento alcuno al **Receiver** da gestire. Questo conduce a una implementazione del **Invoker** completamente svincolata da qualche particolare tipologia di **Receiver**. In questo modo il **Invoker** potrebbe gestire non solo TV, ma qualunque altra tipologia di oggetto (ad es. Hi-Fi, condizionatori d'aria, ecc.), bastando soltanto la disponibilità dei **ConcreteCommnad**. Il costo è che la gestione del riferimento alla TV viene ripetuto in ogni specifica implementazione di comando.

Se invece di una interfaccia `Command` se avesse definito una classe astratta `Command` che gestisse il riferimento alla `TV`, in modo che i **ConcreteCommand** la potessero ereditare (come apparentemente risulta più comodo), il **Invoker** risulterebbe costretto ad agire soltanto su quella tipologia di TV.

Una soluzione per il problema descritto potrebbe essere l'inserimento tra l'interfaccia Command e i **ConcreteCommand**, di una classe astratta (che implementa l'interfaccia `Command`) e che fornisce le operazioni comuni a tutti i **ConcreteCommand**.

Esecuzione dell'esempio

```
c:\Design Patterns\Behavioral\Command >java CommandExample

Select a button to press or 9 to exit
1.- POWER
2.- ARROW UP
3.- ARROW DOWN
4.- NUMERIC KEYPAD
1
The TV is ON.

Select a button to press or 9 to exit
1.- POWER
2.- ARROW UP
3.- ARROW DOWN
4.- NUMERIC KEYPAD
4
Enter the channel number:
12
Current channel is 12

Select a button to press or 9 to exit
1.- POWER
2.- ARROW UP
3.- ARROW DOWN
4.- NUMERIC KEYPAD
2
Current volume level is 1

Select a button to press or 9 to exit
1.- POWER
2.- ARROW UP
3.- ARROW DOWN
4.- NUMERIC KEYPAD
2
Current volume level is 2

Select a button to press or 9 to exit
1.- POWER
2.- ARROW UP
3.- ARROW DOWN
4.- NUMERIC KEYPAD
3
Current volume level is 1
```

14.6. Osservazioni sull'implementazione in Java

Il **Command**, se non fornisce implementazione di default, può diventare una interfaccia, invece della classe astratta suggerita nella proposta originale del pattern.

Paranj [13] offre un altro interessante esempio di applicazione di questo pattern in Java.

15. Interpreter

(GoF pag. 243)

15.1. Descrizione

Dato un linguaggio, definisce una rappresentazione dalla sua grammatica insieme ad un interprete che utilizza questa rappresentazione per l'interpretazione delle espressioni in quel linguaggio.

15.2. Esempio

Si desidera costruire un applicativo per la valutazione di espressioni logiche[10], in diversi contesti, dove un contesto rappresenta un insieme di valori assegnati alle variabili presenti nell'espressione. Una esempio di espressione logica, è:

(*true* AND p) OR (q AND NOT p)

Nell'espressione precedente **p** e **q** sono variabili logiche, ***true*** è una costante e **AND**, **OR** e **NOT** sono gli operatori. Il risultato di questa espressione si presenta di seguito per ogni possibile combinazione di **p** e **q**:

Contesto	p	q	Risultato
1	*true*	*true*	*true*
2	*true*	*false*	*true*
3	*false*	*true*	*true*
4	*false*	*false*	*false*

La grammatica per la costruzione di espressioni logiche da gestire in questo applicativo è la seguente:

BooleanExpression ::= VariableExpression | Constant | OrExpression | AndExpression | NotExpression | '(' BooleanExpression ')'
AndExpression ::= BooleanExpression 'AND' BooleanExpression
OrExpression ::= BooleanExpression 'OR' BooleanExpression
NotExpression ::= 'NOT' BooleanExpression
Constant ::= 'true' | 'false'
VariableExpression ::= 'a' | 'b' | … | 'x' | 'y' | 'z'

Si vuole stabilire un meccanismo per rappresentare e interpretare le espressioni logiche secondo la grammatica specificata.

15.3. Descrizione della soluzione offerta dal Pattern

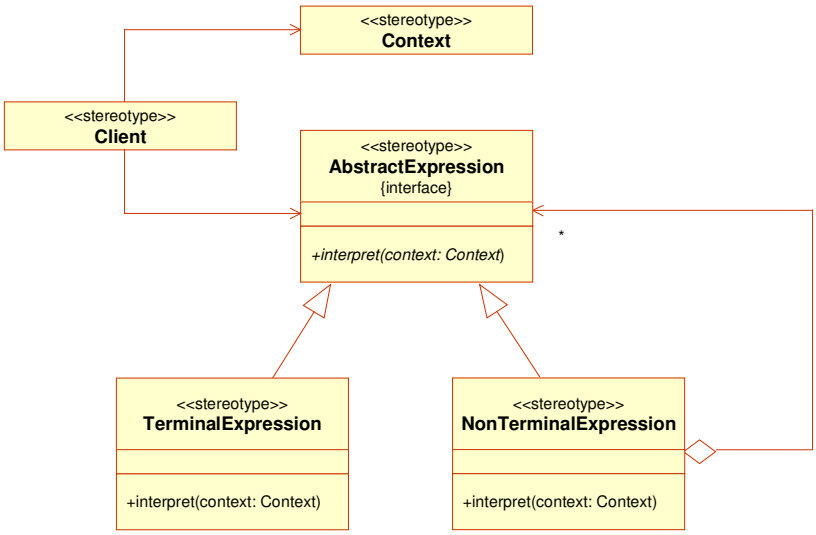
Il pattern "Interpreter" utilizza una classe per rappresentare ogni regola grammaticale. Queste classi, che implementano la stessa interfaccia (AbstractExpression), possono essere espressioni finali (TerminalExpression), oppure espressioni composte da altre espressioni

[10] Questa è una versione leggermente semplificata dell'esempio proposto dai GoF, riguardante l'utilizo di questo pattern in C++.

(NonTerminalExpression). I simboli del lato destro di ogni regola, invece, sono variabili di istanza delle classi.

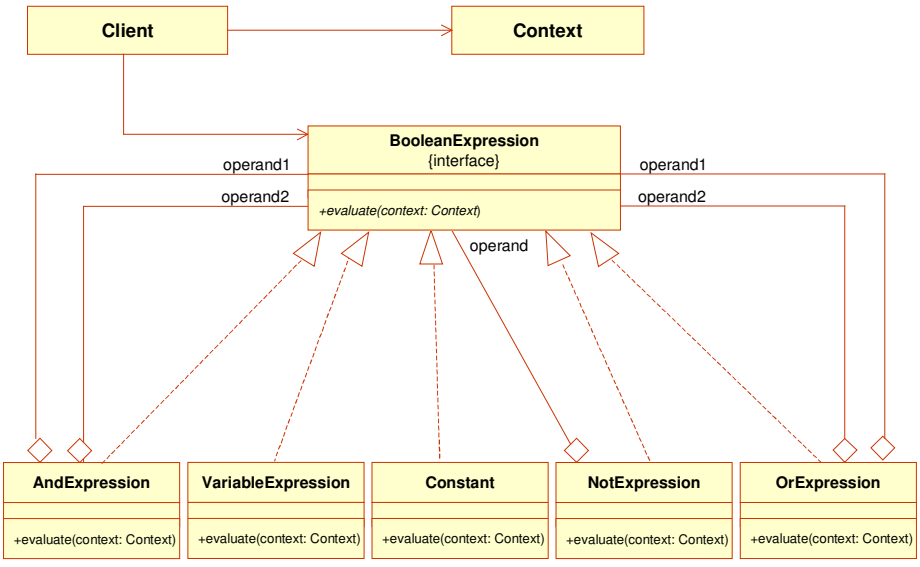
Ogni espressione definita da questa grammatica è rappresentata da un albero di sintassi astratto. L'interpretazione di queste espressioni viene eseguita dall'operazione "interpret" implementata in tutte le classi, che risulta applicata sul contesto di applicazione dell'interprete (Context), dove sono presenti i dati di input.

15.4. Struttura del Pattern



15.5. Applicazione del modello

Schema del modello



Partecipanti

- **AbstractExpression**: interfaccia BooleanExpression.

  - Dichiara una interfaccia contenente l'operazione d'interpretazione ("evaluate", in questo caso), che è comune a tutti i nodi nell'albero astratto di sintassi.

- **TerminalExpression**: classi VariableExpression e Constant.
  - Implementano l'operazione di interpretazione, specificata nell'**AbstractExpression** e associata a espressioni finali della grammatica.
- **NonTerminalExpression**: classi AndExpression, OrExpression e NotExpression.
  - Ogni classe rappresenta una regola della grammatica.
  - Gestisce riferimenti di tipo **AbstractExpression** per ogni simbolo componente .
  - Implementano l'operazione di interpretazione, specificata nell'**AbstractExpression** e associata a espressioni non finali della grammatica. Tipicamente questo consiste in un richiamo alle operazioni d'interpretazioni presenti nei propri componenti.
- **Context**: classe Context.
  - Contiene l'informazione globale sulla quale deve agire l'interprete.
- **Client**: applicazione InterpreterExample.
  - Costruisce l'albero astratto di sintassi, che rappresenta una particolare espressione nel linguaggio che la grammatica definisce.
  - Invoca l'operazione di interpretazione.

Implementazione

Si presenta per prima la classe `InterpreterExample`, che corrisponde all'applicativo che deve valutare una particolare espressione logica:

```
public class InterpreterExample {

 public static void main( String[] args ) {


   VariableExpression p = new VariableExpression( "p" );
   VariableExpression q = new VariableExpression( "q" );

   // Expression: "(true AND p) OR ( q AND NOT p )"
   BooleanExpression expr = new OrExpression(
                             new AndExpression( new Constant(true), p ),
                             new AndExpression(q, new NotExpression(p))
                            );

   Context context = new Context();

   context.assign( p, true );
   context.assign( q, true );
   System.out.println( "(p=true,q=true) The result is: "
                                          + expr.evaluate( context ) );

   context.assign( p, true );
   context.assign( q, false );
   System.out.println( "(p=true,q=false) The result is: "
                                          + expr.evaluate( context ) );
```

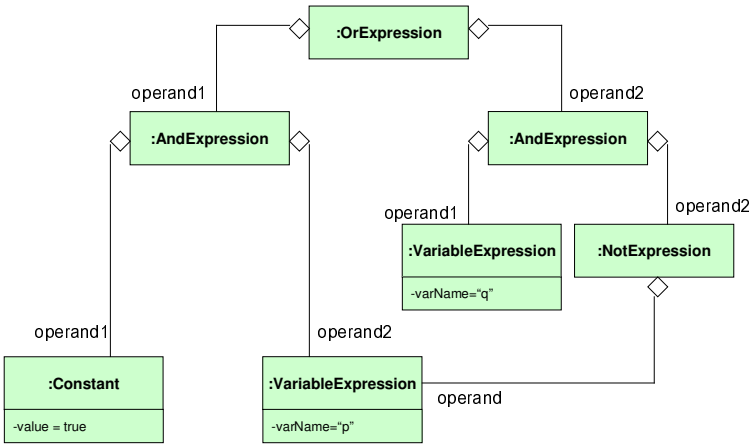
```
    context.assign( p, false );
    context.assign( q, true );
    System.out.println( "(p=false,q=true) The result is: "
                                          + expr.evaluate( context ) );

    context.assign( p, false );
    context.assign( q, false );
    System.out.println( "(p=false,q=false) The result is: "
                                          + expr.evaluate( context ) );

  }

}
```

La classe `InterpreterExample` è anche il **Client** del modello, che crea l'albero astratto di sintassi tramite l'istruzione:

```
    BooleanExpression expr = new OrExpression(
                              new AndExpression( new Constant(true), p ),
                              new AndExpression(q, new NotExpression(p))
                             );
```

L'albero di sintassi derivato da questa espressione si offre nella seguente figura:

Albero di sintassi per l'espressione: (*true* AND p) OR (q AND NOT p)



L'albero di sintassi verrà applicato sul **Context**, rappresentato dalla classe omonima. Questa classe gestisce tramite una Hashtable la lista di variabili con i valori booleani associati, e fornisce due metodi: `assign`, che serve a settare un valore booleano a una particolare variabile, e `lookup`, che consente agli utenti del `Context`, in particolare agli oggetti `VariableExpression`, ricavare il valore booleano definito per le diverse variabili.

```
import java.util.Hashtable;
public class Context {

  private Hashtable vars;

  public Context() {
      vars = new Hashtable();
  }

  public void assign(VariableExpression var, boolean value ) {
      vars.put( var.getVarName() , new Boolean( value ) );
  }

  public boolean lookup( String varName ) {
      return ( (Boolean) vars.get( varName ) ).booleanValue();
```

```
  }

}
```

L'interfaccia `BooleanExpression` (**AbstractExpression**) specifica il metodo evacuate, che ogni tipo di espressione dovrà implementare:

```
public interface BooleanExpression {

  public boolean evaluate( Context context );

}
```

La classe `Constant` è una **TerminalExpression**, che immagazzina un valore booleano da usare come parte delle espressioni. Il metodo `evaluate`, semplicemente restituisce il proprio valore:

```
public class Constant implements BooleanExpression {

      boolean value;

      public Constant(boolean val) {
             value = val;
      }

      public boolean evaluate(Context context) {
             return value;
      }

}
```

La classe `VariableExpression` è un'altra **TerminalExpression** che serve per rappresentare le variabili presenti nell'espressione logica da valutare. Ogni oggetto di questa classe immagazzina il nome di una singola variabile. Il metodo `evaluate` restituisce il valore della variabile, il quale non è presente nell'istanza di `VariableExpression`, ma viene preso dall'oggetto `Context`, tramite il suo metodo `lookup`:

```
public class VariableExpression implements BooleanExpression {

      private String varName;

      public VariableExpression(String name) {
             varName = name;
      }

      public boolean evaluate(Context context) {
             return context.lookup( varName );
      }

      public String getVarName() {
             return varName;
      }
}
```

La classe `AndExpression` è una **NonTerminalExpression** che serve per costruire le operazioni di AND logico. Gli oggetti di questa classe possiedono riferimenti verso le due espressioni (terminali o non terminali) che corrispondono agli operandi. Il metodo `evaluate` richiede la propria valutazione a entrambi operandi, e i valori ottenuti applica l'operatore AND (`&&`), restituendo il risultato ottenuto:

```
public class AndExpression implements BooleanExpression{

    BooleanExpression operand1, operand2;

    public AndExpression(BooleanExpression op1, BooleanExpression op2) {
             operand1 = op1;
             operand2 = op2;
```

```
    }

    public boolean evaluate(Context context) {
             return operand1.evaluate(context) &&
                     operand2.evaluate(context);
    }

}
```

La classe `OrExpression` è molto simile alla classe `AndExpression`, essendo l'unica differenza che il metodo `evaluate` viene valutato combinando i valori degli operandi tramite l'operatore OR (`||`).

```
public class OrExpression implements BooleanExpression{

     BooleanExpression operand1, operand2;

     public OrExpression(BooleanExpression op1, BooleanExpression op2) {
             operand1 = op1;
             operand2 = op2;
     }

     public boolean evaluate(Context context) {
             return operand1.evaluate(context) ||
             operand2.evaluate(context);
     }

}
```

Finalmente, la classe `NotExpression` è l'altra **NonTerminalExpression**, che essendo di natura unaria, agisce soltanto su un singolo operando. Il metodo `evaluate` restituisce il valore negato dell'operando:

```
public class NotExpression implements BooleanExpression {

      BooleanExpression operand;

      public NotExpression(BooleanExpression op) {
             operand = op;
      }

      public boolean evaluate(Context context) {
             return ! operand.evaluate(context);
      }

}
```

Osservazioni sull'esempio

In questo esempio la costruzione dell'albero di sintassi è stato cablato come codice interno del **Client**. Questo albero potrebbe essere costruito dinamicamente tramite la parisifica di una espressione.

Esecuzione dell'esempio

```
c:\Design Patterns\Behavioral\Interpreter >java InterpreterExample

Expression: '(true AND p) OR ( q AND NOT p )'

(p=true,q=true) The result is: true
(p=false,q=true) The result is: true
(p=true,q=false) The result is: true
(p=false,q=false) The result is: false
```

15.6. Osservazioni sull'implementazione in Java

Non ci sono considerazioni particolari riguardanti l'implementazione di questo pattern in Java.

16. Iterator

(GoF pag. 257)

16.1. Descrizione

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

16.2. Esempio

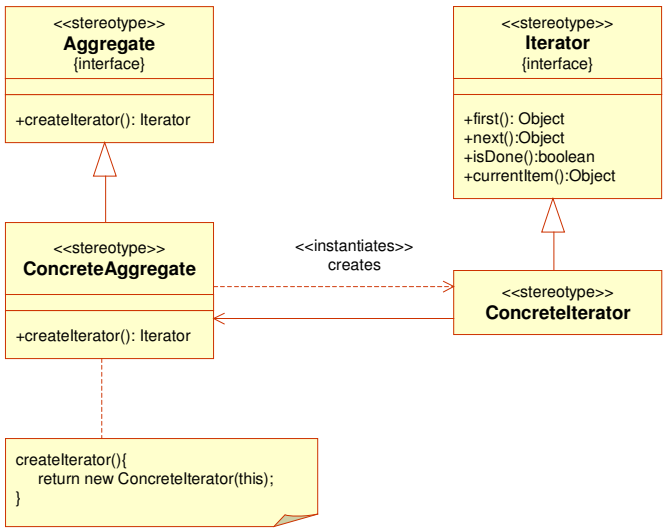
Il percorso di un viaggiatore è rappresentato come una collezione ordinata di oggetti, dove ogni oggetto rappresenta un luogo visitato. La collezione può essere implementata in base a *array*, una *linked list* o qualunque altra struttura.

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.

16.3. Descrizione della soluzione offerta dal pattern

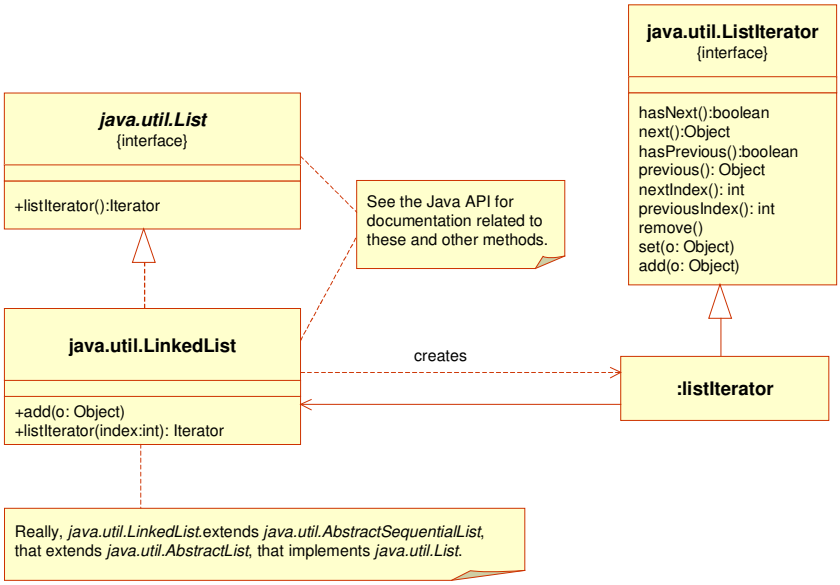
Il pattern "*Iterator*" suggerisce l'implementazione di un oggetto che consenta l'acceso e percorso della collezione, e che fornisca una interfaccia standard verso chi è interessato a percorrerla e ad accede agli elementi.

16.4. Struttura del Pattern



16.5. Applicazione del pattern

Schema del modello



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.
- **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()) );
  }

}
```

Osservazioni sull'esempio

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

Esecuzione dell'esempio

```
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
```

16.6. 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.

17. Mediator

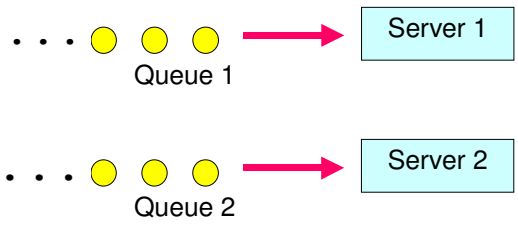
(GoF pag. 273)

17.1. Descrizione

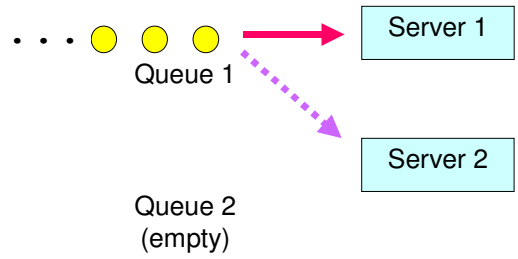
Definisce un oggetto che incapsula il modo di interagire di un gruppo d'oggetti, consentendo il disaccoppiamento tra questi, in forma tale di poter variare facilmente le interazioni fra di loro.

17.2. Esempio

Un simulatore di un sistema bancario consente di creare diverse configurazioni di cassieri (teller) e clienti in coda (queue). Ad esempio, una delle configurazioni possibili si struttura in base a due cassieri che servono soltanto i propri clienti in coda:



Un'altra configurazione, invece, può ammettere il servizio di un cassiere ai clienti della coda dell'altro server, se la propria è vuota:



Questo simulatore potrebbe gestire, anche, altri modi d'interazione tra questi partecipanti.

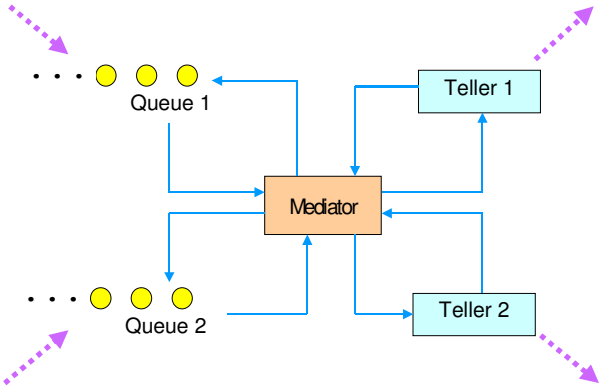
Una progettazione semplicista delle classi del simulatore porterebbe a stabilire dei legami tra le configurazioni possibili e l'implementazione dei componenti. Per esempio, per la gestione della seconda modalità di funzionamento, appena illustrata, il servizio del Teller, al suo interno potrebbe avere codice del tipo:

```
If( mode == SERVE_ONLY_OWN_QUEUE )
   if( !ownQueue.isEmpty() )
     OwnQueue.initService();
else
   if( ownQueue.isEmty() && ! otherServerQueue.isEmpty() )
       otherServerQueue.initService();
...
```

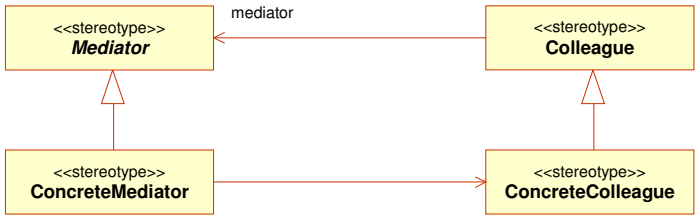
Questo codice rende più difficile riutilizzare le classi per creare nuove configurazioni del sistema, dato che le interazioni possibili devono essere previste in tempo di sviluppo di ogni singolo componente. In questo senso sarebbe utile una progettazione conducente ad un modello di interazione più flessibile tra le diverse classi.

17.3. Descrizione della soluzione offerta dal pattern

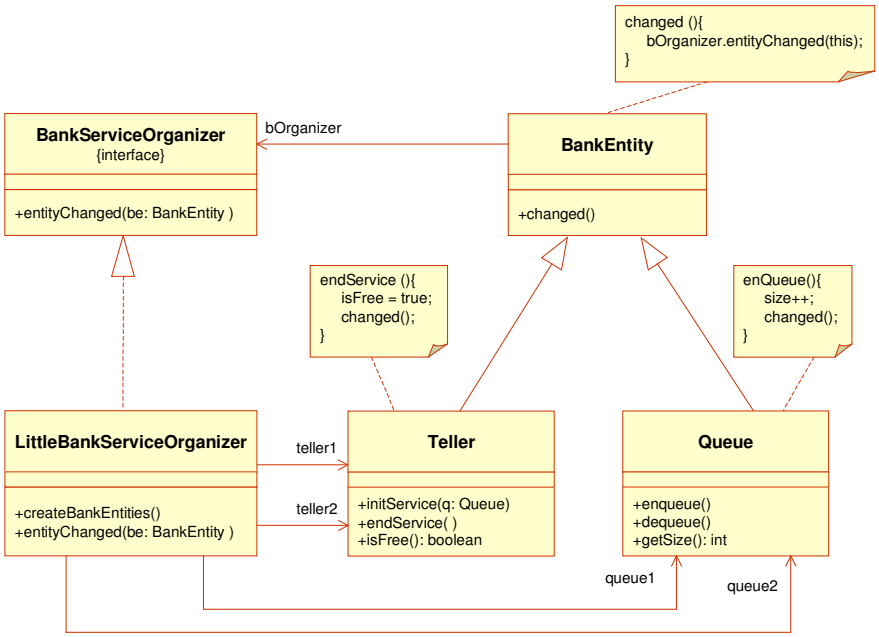
Questo pattern propone la creazione di incapsulare il comportamento collettivo delle diversi classi componeti il sistema (colleagues), tramite una classe separata denominata *Mediator*. Il Mediator diventa l'agente d'intermediazione tra i diversi oggetti, i quali soltanto devono interfacciarsi con esso, riducendo il numero di interconnessioni.



17.4. Struttura del Pattern



Schema del modello



Partecipanti

- **Mediator**: interfaccia BankServiceOrganizer.
  - Specifica una interfaccia per la comunicazione da parte dei **Colleagues**.
- **ConcreteMediator**: classe LittleBankServiceOrganizer.
  - Implementa il comportamento cooperativo tramite la coordinazione dei **Colleagues**.
  - Possiede riferimenti verso i **Colleagues**.
- **Colleague**: classe BankEntity.
  - Possiede un riferimento al **Mediator**.
  - Implementa un metodo di notifica di eventi al **Mediator**.
- **ConcreteColleague**: classi Teller e Queue.
  - Comunica gli eventi al **Mediator** invece di comunicarli ad altri **Colleagues**.

Implementazione

Si definisce la classe astratta `BankEntity` (**Colleague**), che serve di base per le classi che rappresentano code e cassieri (ConcreteColleagues). `BankEntity` include le operazioni necessarie per registrare il **Mediator** con il quale dovrà interagire, e anche include il metodo di notifica al **Mediator** dei cambi di stati rilevanti

```
public abstract class BankEntity {

    BankServiceOrganizer bOrganizer;

    public BankEntity( BankServiceOrganizer bso ) {
```

```
        bOrganizer = bso;
    }

    public void changed() {
        bOrganizer.entityChanged( this );
    }

}
```

Qualunque **ConcreteMediator** è visto da ogni tipo di **Colleague** tramite l'interfaccia `BankServiceOrganizer` (**Mediator**). Questa interfaccia dichiara il metodo `entityChanged` che sarà invocato dai **Colleagues** per notificare il **Mediator** i propri cambiamenti di stato:

```
public interface BankServiceOrganizer {
    public void entityChanged( BankEntity be );
}
```

La classe `Teller` rappresenta i **Collegues** che agiscono come server del sistema. Si noti che il servizio del `Teller` non fa altro che togliere un oggetto di una coda. Gli eventi riguardanti l'operazione di un `Teller` sono l'inizio del servizio e il termine del servizio. Ogni inizio di servizio sarà scattato dal **Mediator**, intanto che un fine di servizio sarà raggiunto tramite l'invocazione del metodo `endService` (da un generatore di eventi). L'aspetto importante è che i fini di servizi saranno notificati al Mediator in modo che questo possa determinare la coda da essere servita:

```
public class Teller extends BankEntity {

    private boolean isFree;

    public Teller( BankServiceOrganizer bso ) {
        super( bso );
        isFree = true;
    }

    public void initService( Queue q ) {
        if( isFree ) {
        q.dequeue();
          isFree = false;
        }
    }

    public void endService( ) {
        isFree = true;
        changed();
    }

    public boolean isFree() {
        return isFree;
    }

}
```

Gli oggetti coda saranno rappresentati da istanze della classe `Queue`. In questa classe il metodo `enqueue`, che rappresenta l'arrivo di un nuovo cliente, fa una notifica al **Mediator**, perché se la coda è originalmente vuota, e al meno un server per quella coda disponibile, il **Mediator** deve richiedere il servizio su di essa.

```
public class Queue extends BankEntity {

    private int size;

    public Queue( BankServiceOrganizer bso ) {
        super( bso );
    }

    public void enqueue( ) {
        size++;
        changed();
    }

    public void dequeue( ) {
        if( size > 0 )
          size--;
    }

    public int getSize() {
        return size;
    }

}
```

`BankServiceOrganizer` rappresenta l'interfaccia del **Mediator**:

```
public interface BankServiceOrganizer {
    public void entityChanged( BankEntity be );
}
```

La classe `LittleBankServiceOrganizer` è un **ConcreteMediator**. Il metodo `createEntities` crea istanze dei colleghi, inviando un riferimento a se stesso ai loro costruttori. Il metodo `entityChanged` include tutta la logica riguardante le operazioni di interazione tra servitori e code:

```
public class LittleBankServiceOrganizer implements BankServiceOrganizer
{

    private Queue queue1, queue2;
    private Teller teller1, teller2;

    public void createBankEntities() {
       queue1  = new Queue( this );
       queue2  = new Queue( this );
       teller1 = new Teller( this );
       teller2 = new Teller( this );
    }

    public void entityChanged( BankEntity entityChanged ) {
       if( entityChanged == teller1 ) {
          if( queue1.getSize() > 0 )
          teller1.takeClientFrom( queue1 );
        else
          if( queue2.getSize() > 0 )
             teller1.takeClientFrom( queue2 );
       } else if( entityChanged == teller2 ) {
          if( queue2.getSize() > 0 )
          teller2.takeClientFrom( queue2 );
        else
          if( queue1.getSize() > 0 )
             teller2.takeClientFrom( queue1 );
       } else if( entityChanged == queue1 ) {
          if( teller1.isFree() )
          teller1.takeClientFrom( queue1 );
        else
          if( teller2.isFree() )
             teller2.takeClientFrom( queue1 );
       } else if( entityChanged == queue2 ) {
          if( teller2.isFree() )
          teller2.takeClientFrom( queue2 );
        else
          if( teller1.isFree() )
```

```
            teller1.takeClientFrom( queue2 );
         }
      }

      public Queue getQueue1() {
         return queue1;
      }

      public Queue getQueue2() {
         return queue2;
      }

      public Teller getTeller1() {
         return teller1;
      }

      public Teller getTeller2() {
         return teller2;
      }

}
```

I metodi `getQueue1`, `getQueue2`, `getTeller1`, `getTeller2` servono soltanto a restituire al simulatore un riferimento ad ogni oggetto, in modo che un generatore di eventi esterno al **Mediator**, possa fare scattare l' arrivo di clienti e la fine di servizio.

Si presenta la classe `LittleBank` che agisce come generatore di eventi esterno, che ai suoi interni crea una istanza del **Mediator**, e che ottiene da eso, riferimenti a code e server.

```
public class LittleBank {

    private Queue  queue1, queue2;
    private Teller teller1, teller2;

    public LittleBank(  ) {
       LittleBankServiceOrganizer lbso =
                                      new LittleBankServiceOrganizer();
       lbso.createBankEntities();
       queue1 = lbso.getQueue1();
       queue2 = lbso.getQueue2();
       teller1 = lbso.getTeller1();
       teller2 = lbso.getTeller2();
    }

    public void customerArriveToQ1() {
        queue1.enqueue();
    }

    public void customerArriveToQ2() {
        queue2.enqueue();
    }

    public void endServiceTeller1() {
        teller1.endService();
    }

    public void endServiceTeller2() {
        teller2.endService();
    }

    public void showStatus() {
      System.out.println( "---------------------------------"  );
      System.out.println( "Teller 1 status: "
                          + ( teller1.isFree() ? "FREE" : "BUSY" ) );
      System.out.println( "Teller 2 status: "
                          + ( teller2.isFree() ? "FREE" : "BUSY" ) );
      System.out.println( "Queue 1 size: " + queue1.getSize() );
      System.out.println( "Queue 2 size: " + queue2.getSize() );
    }
}
```

Finalmente, il codice del `main` dell'esempio:

```
public class MediatorExample {

    public static void main( String[] arg ) {

       LittleBank bank = new LittleBank( );

       bank.showStatus();
       System.out.println( "A customer arrives to queue 1" );
       bank.customerArriveToQ1();
       bank.showStatus();
       System.out.println( "A customer arrives to queue 1" );
       bank.customerArriveToQ1();
       bank.showStatus();
       System.out.println( "A customer arrives to queue 1" );
       bank.customerArriveToQ1();
       bank.showStatus();
       System.out.println( "A customer arrives to queue 2" );
       bank.customerArriveToQ2();
       bank.showStatus();
       System.out.println( "End of service teller 1" );
       bank.endServiceTeller1();
       bank.showStatus();
       System.out.println( "End of service teller 2" );
       bank.endServiceTeller2();
       bank.showStatus();
       System.out.println( "End of service teller 2" );
       bank.endServiceTeller2();
       bank.showStatus();

    }

}
```

Osservazioni sull'esempio

La classe `LittleBank` non forma parte dell'architettura del pattern *Mediator*. La sua implementazione è stata necessaria in questo esempio per incapsulare la gestione degli eventi arrivo e uscita di clienti. Questi eventi aggiornano lo stato di code e server i cui cambiamenti fanno evolvere la simulazione del sistema.

Esecuzione dell'esempio

```
C:\Design Patterns\Behavioral\Mediator>java MediatorExample

---------------------------------
Teller 1 status: FREE
Teller 2 status: FREE
Queue 1 size: 0
Queue 2 size: 0
A customer arrives to queue 1
---------------------------------
Teller 1 status: BUSY
Teller 2 status: FREE
Queue 1 size: 0
Queue 2 size: 0
A customer arrives to queue 1
---------------------------------
Teller 1 status: BUSY
Teller 2 status: BUSY
Queue 1 size: 0
Queue 2 size: 0
A customer arrives to queue 1
---------------------------------
Teller 1 status: BUSY
Teller 2 status: BUSY
Queue 1 size: 1
Queue 2 size: 0
A customer arrives to queue 2
---------------------------------
Teller 1 status: BUSY
Teller 2 status: BUSY
Queue 1 size: 1
Queue 2 size: 1
End of service teller 1
---------------------------------
Teller 1 status: BUSY
Teller 2 status: BUSY
Queue 1 size: 0
Queue 2 size: 1
End of service teller 2
---------------------------------
Teller 1 status: BUSY
Teller 2 status: BUSY
Queue 1 size: 0
Queue 2 size: 0
End of service teller 2
---------------------------------
Teller 1 status: BUSY
Teller 2 status: FREE
Queue 1 size: 0
Queue 2 size: 0
```

17.5. Osservazioni sull'implementazione in Java

Dato che il **Mediator** dichiara ma non implementa operazioni, questo viene specificato come una interfaccia in Java.

18. Memento

(GoF pag. 283)

18.1. Descrizione

Senza violare l'incapsulazione di un oggetto, cattura e porta fuori di sé il suo stato interno, in modo che questo stato possa essere posteriormente ristabilito.

18.2. Esempio

Un viaggiatore, in ogni tappa del suo viaggio, decide casualmente il prossimo destino. Quando il viaggiatore arriva al destino determinato, decide se il luogo è del suo gusto o meno. Nel primo caso, determina per un nuovo destino, nel secondo, invece, ritorna al luogo da dove era partito.

Si noti che l'algoritmo di viaggio non consente di determinare le condizioni di partenza (per esempio, la distanza effettiva percorsa a quel punto, e la lunghezza dell'ultimo tratto camminato) in base alle condizioni di arrivo, dato che il viaggio è deciso casualmente.

Un semplice modo per gestire il ritorno alle condizioni di partenza segue l'approccio di esternalizzare i valori delle variabili che rappresentano lo stato, vale dire, copiarle in variabili esterne per eventualmente poter ripristinare il loro valore. Questa però può essere una soluzione poco adeguata dal punto di vista della programmazione orientata ad oggetti, poiché si vulnera l'incapsulazione dell'oggetto, rendendo visibili i suoi dettagli implementativi.

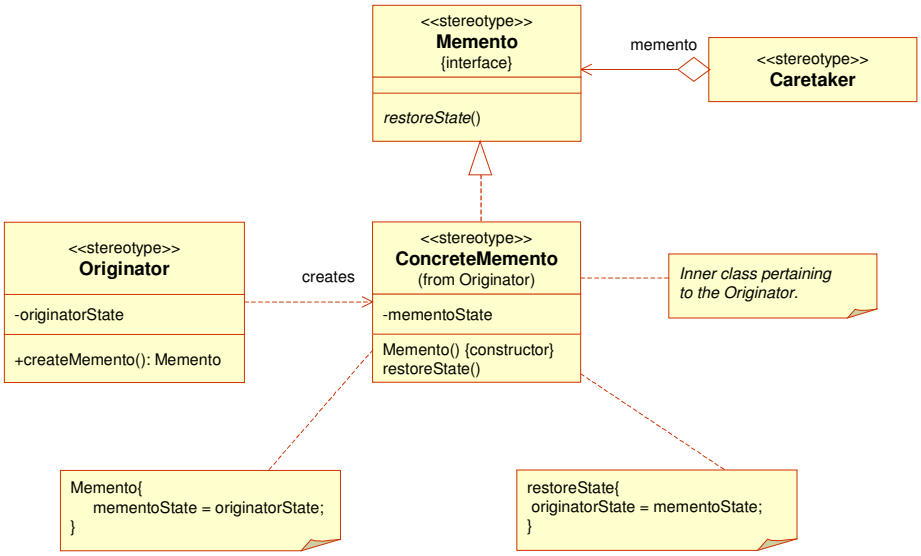
Il problema è trovare un meccanismo per una esternalizzazione dello stato, in modo di poter supportare operazioni di ripristino, senza esporre i dettagli di implementazione.

18.3. Descrizione della soluzione offerta dal pattern

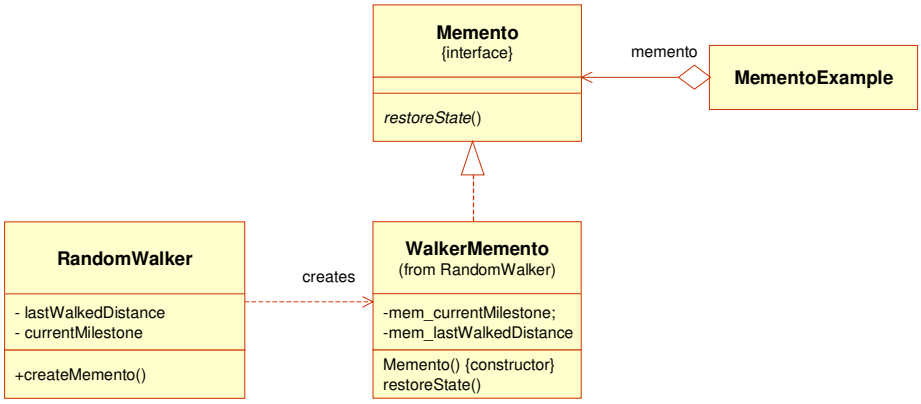
Il "*Memento*" è un pattern in grado di soddisfare i requisiti esposti nella sezione precedente. Lo stato dell'oggetto che si vuole salvare (Originator) è immagazzinato in un oggetto (Memento) che è portato fuori, e ritenuto esternamente da un altro oggetto (Caretaker). Quando si vuole restituire uno stato immagazzinato dell'Originator, è invocato un suo metodo di ripristino che agisce in base a un Memento che contiene quel particolare stato.

18.4. Struttura del Pattern

Si offre una struttura originale del pattern, che risulta leggermente diversa a quella proposta da GoF, come si spiega poi nella descrizione del codice.



18.5. Applicazione del pattern



Schema del modello

Partecipanti

- **Memento**: interfaccia Memento.
  - Specifica l'interfaccia che i **ConcreteMemento** espongono al **Caretaker**. Questa interfaccia impedisce l'accesso alle proprie variabili, e fornisce soltanto l'operazione di restituzione dello stato.
- **ConcreteMemento**: Inner class WalkerMemento, appartenete alla classe RandomWalker.
  - Implementa l'interfaccia **Memento**.
  - Immagazzina lo stato interno di un oggetto **Originator**.
- **Originator**: classe RandomWalker.

  - Crea una istanza di **Memento** come *inner class*, contenente la fotografia del proprio stato corrente (o di una porzione di esso).
  - Utilizza il **Memento** per restituire il proprio stato interno.

- **CareTaker**: applicazione MementoExample.
  - Conserva il **Memento**.
  - Mai esamina il contenuto del **Memento**.

Implementazione

L'implementazione comune suggerita da diversi autori, si basa nell'utilizzo della visibilità di package come meccanismo di implementazione della dualità d'interfaccia, aperta verso l'**Originator** e chiusa verso il **Caretaker**, che il **Memento** deve offrire. Seguendo questo approccio **Originator** e **Memento** devono appartenere allo steso package e avere definito le variabili di stato con la visibilità di default (`package`). Si veda [4] per una esemplificazione.

Questo testo offre invece una implementazione originale del pattern Memento, basata sulle classi nidificate (*nested classes*), che risulta più semplice e vantaggiosa da gestire.

Si presenta in primo luogo la classe `RandomWalker` che rappresenta il viaggiatore. Ogni oggetto di questa classe ha un attributo che contiene la distanza effettiva camminata dal viaggiatore (`currentMilestone`, quella che lo separa dal punto di partenza) e la lunghezza dell'ultimo tratto percorso (`lastWalkedDistance`). Ogni volta che è invocato il metodo `randomWalk` si genera in modo casuale la distanza del percorso della tratta e si modifica lo stato dell'oggetto[11].

```
public class RandomWalker {

  private int currentMilestone;
  private int lastWalkedDistance;
  private RandomWalker randomWalker;

  public RandomWalker() {
     randomWalker = this;
  }

  public void randomWalk() {
     lastWalkedDistance = (int) ( Math.random() * 100 );
     currentMilestone = currentMilestone + lastWalkedDistance;
  }

  public int getCurrentMilestone() {
     return currentMilestone;
  }

  public int getLastWalkedDistance() {
     return lastWalkedDistance;
  }

  public Memento createMemento( ) {
     return new WalkerMemento( );
  }
```

[11] Si noti che con gli attributi presenti nel Viaggiatore, lo si può riportare allo stato di origine di una tratta solo parzialmente: alla `currentMilestone` si potrebbe restare la `lastWalkedDistance` per trovare la `currentMilestone` originale, ma non è possibile ripristinare la `lastWalkedDistance` originale.

```
  class WalkerMemento implements Memento{
    private int mem_currentMilestone;
    private int mem_lastWalkedDistance;
    public WalkerMemento() {
      mem_currentMilestone   = currentMilestone;
      mem_lastWalkedDistance = lastWalkedDistance;
    }
    public void restoreState()  {
      currentMilestone   = mem_currentMilestone;
      lastWalkedDistance = mem_lastWalkedDistance;
    }

  } //End of class WalkerMemento

} //End of class RandomWalker
```

Si noti che la classe `RandomWalker` contiene come membro la dichiarazione della classe `WalkerMemento`. Ogni volta che si vuole creare un **Memento**, è invocato il metodo `createMemento`, il quale istanza un `WalkerMemento` che registra nelle proprie variabili d'istanza (private) lo stato dell'**Originator**. Si noti in forma particolare che sebbene gli attributi dell'**Originator** sono privati, cioè non accessibili da nessun'altra classe ordinaria, i loro valori sì lo sono dalla classe nidificata `WalkerMemento`.

Da un'altra parte, il ciclo di vita del `WalkerMemento` è strettamente legato all'oggetto `RandomWalker` al quale appartiene, così che se questo è cancellato (garbage collected) il **Memento** scompare con lui.

Il `WalkerMemento` implementa l'interfaccia `Memento`, che ha un unico metodo, `restoreState`. Questo metodo ripristina lo stato dell'oggetto registrato nel Memento.

```
public interface Memento {

   public void restoreState();

}
```

Il **Caretaker** vede il **Memento** tramite questa interfaccia, perciò sarà soltanto in grado di chiedere la restituzione dello stato dell'**Originator**, senza dover indicare al **Memento** su quale **Originator** agire (dato che una oggetto appartenente a una classe nidificata può avere visibilità soltanto sulle variabili dello stesso oggetto a cui appartiene). Questa implementazione consente una gestione più semplicistica della originalmente proposta da GoF, dove il Caretaker deve indicare, come parametro del metodo di restituzione dello stato, il corretto Originator su cui agire.

In questo esempio, il **Caretaker** è la stessa applicazione (`MementoExample`), che mantiene soltanto un unico **Memento** dello stato del viaggiatore.

```
public class MementoExample {

 public static void main (String[] arg) {

    RandomWalker luke = new RandomWalker();
    // Creates a Memento that saves the original state
```

```
    Memento tripStop  = luke.createMemento();

    for(int i=1; i<=4 ;i++) {
       System.out.println( "Starting trip..." );
       luke.randomWalk();
       whereIs( luke );
       System.out.println( "Do you like this place?" );
       if(Math.random() < .4) {
         System.out.println( "-No!" );
         // Restores the last saved state
         tripStop.restoreState();
         whereIs( luke );
       }else {
         System.out.println( "-Yes!" );
         // Creates a new Memento to save the new state
         tripStop = luke.createMemento();
       }
    }
    System.out.println( "You reach the km "+luke.getCurrentMilestone());
 }

  public static void whereIs( RandomWalker rw ) {
    System.out.print  ( "You are now stopped at km " +
                         rw.getCurrentMilestone()+ ". " );
    System.out.println( "This place is "+ rw.getLastWalkedDistance()
                        + " kms far from your last stop. ");
  }

}
```

Osservazioni sull'esempio

In questo esempio è l'applicazione quella che ha il ruolo del **Caretaker**.

Il **Caretacker** presentato conserva soltanto un unico **Memento** dell'**Originator**, ma facilmente potrebbe progettarsi la conservazione di più stati, ad esempio, tramite una struttura di *stack*, che consentirebbe di fornire un meccanismo di *undo* attraverso tutti i cambiamenti di stato dell'**Originator**.

Esecuzione dell'esempio

```
C:\Design Patterns\Behavioral\Memento>java MementoExample

Starting trip...
You are now stopped at km 14. This place is 14 kms far from your last
stop.
Do you like this place?
-Yes!

Starting trip...
You are now stopped at km 47. This place is 33 kms far from your last
stop.
Do you like this place?
-No!
You are now stopped at km 14. This place is 14 kms far from your last
stop.

Starting trip...
You are now stopped at km 25. This place is 11 kms far from your last
stop.
Do you like this place?
-Yes!

Starting trip...
You are now stopped at km 116. This place is 91 kms far from your last
stop.
Do you like this place?
-No!
You are now stopped at km 25. This place is 11 kms far from your last
stop.
You reach the km 25
```

18.6. Osservazioni sull'implementazione in Java

L'implementazione offerta del Memento trae beneficio dalle classi nidificate di Java, consentendo la costruzione di un **Memento** di architettura leggermente diversa da quello proposto dai GoF. La differenza sostanziale si trova nel fatto che il **Memento** sia strettamente legato all'istanza di **Originator** che lo ha creato, con le seguenti conseguenze:

- Il metodo `restoreState` del **Memento** non richiede un parametro che indichi quale **Originator** ripristinare.
- Le variabili di istanza dell'**Originator** possono essere private. Nell'implementazione basata su package queste devono essere protected.
- Quando l'**Originator** è distrutto, i loro **Memenți** sono anche distrutti, il **Caretaker** deve tener cura di questo fatto per non sollevare `NullPointerException`s al momento di voler ripristinare lo stato di un oggetto di una variabile che in momento referenziava un Memento, ma che poi dopo automaticamente è distrutto.
- Un **Memento** implementato in questo modo non può restituire lo stato di un Originator in un altro oggetto della stessa classe.

Si vuol rendere noto che un altro modo (un poco meno efficiente) d'implementare la funzionalità di questo design pattern sarebbe tramite la serializzazione dello stato dell'**Originator**.

19. Observer

(GoF pag. 293)

19.1. 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.

19.2. 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.

19.3. Descrizione della soluzione offerta dal pattern

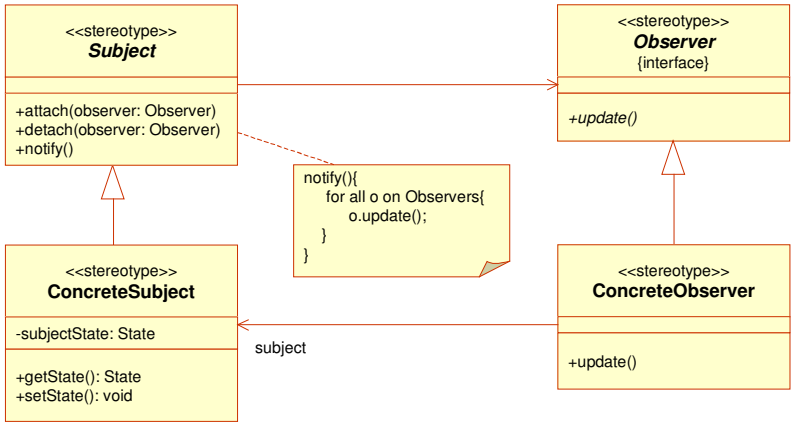
Il pattern "Observer" assegna all''oggetto monitorato (Subject) il ruolo di registrare ai suoi interni un riferimento agli altri oggetti che devono essere avvisati (ConcreteObservers) degli eventi del Subject, e notificarli tramite l'invocazione a un loro metodo, presente nella interfaccia che devono 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.

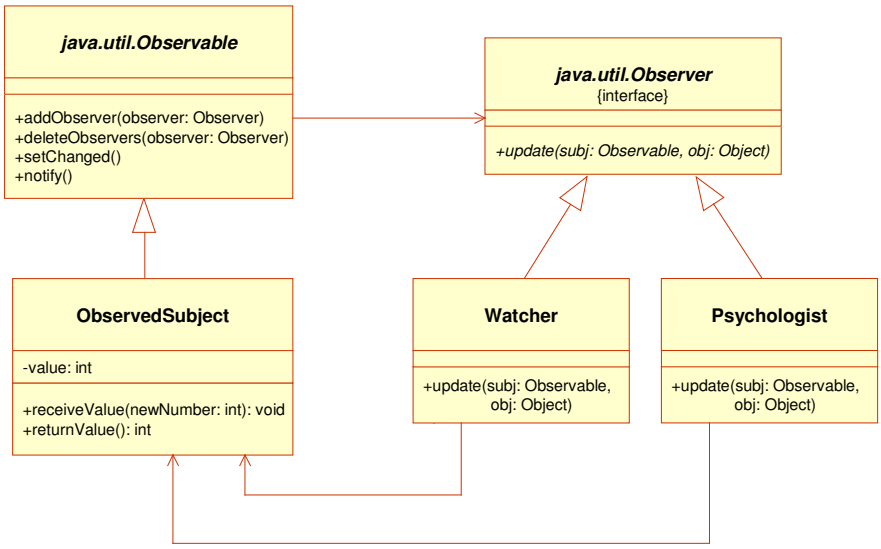
Un'altra versione del pattern Observer, esteso con una gestione più completa degli eventi, è implementato dentro l'ambiente di sviluppo G++ [16]. In questo modello è consentita la registrazione del Observer presso il Subject, indicando addizionalmente il tipo di evento davanti al quale l'Observer deve essere notificato, e la funzione del Observer da invocare.

19.4. Struttura del pattern



19.5. Applicazione del pattern

Schema del modello



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 ).returnValue() + ".");
      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) {

       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.

Esecuzione dell'esempio

```
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.
```

19.6. 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].

20. State

(GoF pag. 305)

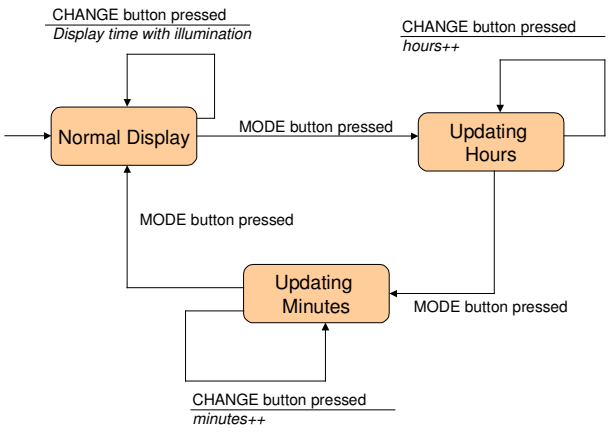
20.1. Descrizione

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

20.2. 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 seguente diagramma di stati 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:

```
operation buttonCHANGEpressed{

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

}
```

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

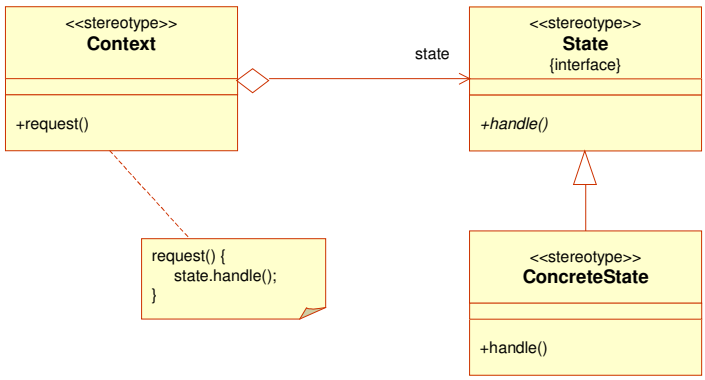
(l'orologio in questo caso), a seconda del proprio stato, perché questo comportamento è spezzato dentro l'insieme di operazioni disponibili.

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.

20.3. Descrizione della soluzione offerta dal Pattern

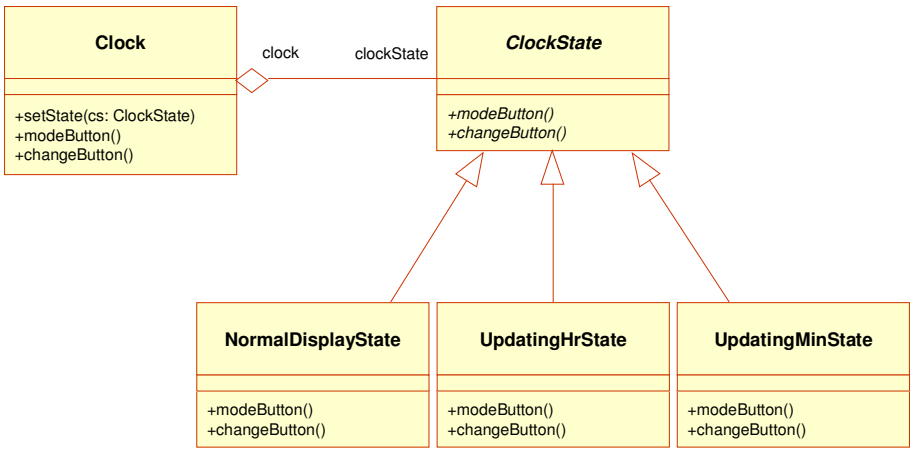
Il pattern "*State*" suggerisce incapsulare, all'interno di una classe, il modo particolare in cui le operazioni di un oggetto (Context) vengono svolte quando lo si trova in quello stato. Ogni classe (ConcreteState) rappresenta un singolo stato possibile del Context e implementa una interfaccia comune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

20.4. Struttura del Pattern



20.5. Applicazione del Pattern

Schema del modello



Partecipanti

- **Context**: classe Clock.
  - 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

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.

Esecuzione dell'esempio

```
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.
```

20.6. Osservazioni sull'implementazione in Java

Non ci sono aspetti particolari da tenere in considerazione.

21. Strategy

(GoF pag. 315)

21.1. 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.

21.2. 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 …
```

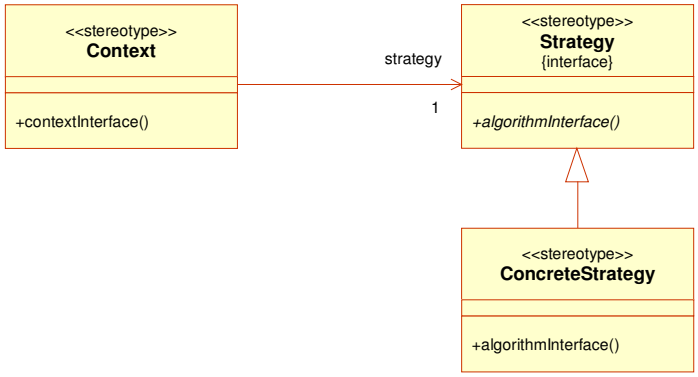
E' anche valido pensare che questi formati potrebbero posteriormente essere sostituiti da altri.

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.

21.3. Descrizione della soluzione offerta dal pattern

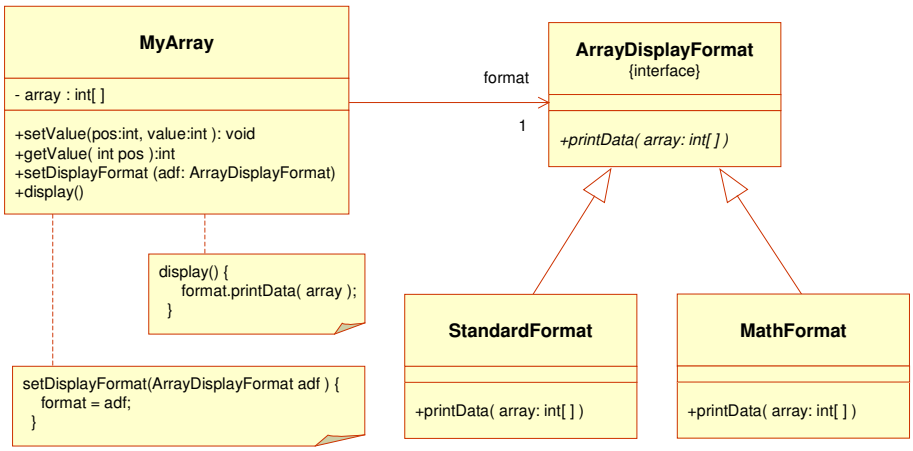
Lo "*Strategy*" pattern suggerisce l'incapsulazione della logica di ogni particolare algoritmo, in apposite classi (ConcreteStrategy) che implementano l'interfaccia che consente agli oggetti MyArray (Context) di interagire con loro. Questa interfaccia deve fornire un accesso efficiente ai dati del Context,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

21.4. Struttura del Pattern



21.5. Applicazione del pattern

Schema del modello



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.
  - 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 `getValue`. 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 …"):

```
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.

Esecuzione dell'esempio

```
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
```

21.6. Osservazioni sull'implementazione in Java

Non ci sono aspetti particolari da tenere in conto.

22. Template method

(GoF pag. 325)

22.1. Descrizione

Specifica la struttura algoritmica di una operazione, differendo l'implementazione di alcuni passi alle sottoclassi. Il Template Method consente ridefinire certi passi di un algoritmo senza cambiare la struttura di esso.

22.2. Esempio

Un sistema di controllo di un particolare magazzino gestisce due tipologie di articoli:

- Articoli generali: per ogni articolo si ha conoscenza dello stock fisico (on-hand), lo stock in transito (in-transit) e le ordini in attesa di arrivo dell'articolo (backorder)[12]. Ogni volta che si richiede un articolo di questo tipo, la quantità richiesta è diminuita dallo stock fisico. Nel caso in cui lo stock fisico non è sufficiente per soddisfare la richiesta, la differenza non coperta viene sommata ai backorder, ma soltanto se questa può essere soddisfatta dal stock in transito, non riservato da altre backorders.
- Articoli ristretti: sono articoli che si consegnano soltanto se lo stock fisico è sufficiente, e non si mantiene traccia né degli articoli in arrivo, né dei backorders. Oltre a questo, il sistema gestisce delle quote che limitano il lotto da richiedere per volta.

Si noti che, astraendo adeguatamente un modello operativo, si riesce a stabilire un meccanismo comune per la gestione di entrambe tipologie di articoli:

```
Algoritmo Ritirare_Items ( quantità )

  begin
     if numero_valido( quantità )
          if puo_essere_soddisfatta( quantità )
             ritirare( quantità )
          else
             write( "stock insufficiente"  )
          endif
     else
          write( "quantità richiesta non valida" )
     endif
  end
```

[12] In un sistema di magazzino i *backorder* rappresentano richieste di articoli che non sono soddisfatte per la mancanza di stock fisico, e che restano in attesa che questi arrivino.

In questo modello, però, le istruzioni in grassetto hanno un'interpretazione particolare per ogni tipologia di articolo:

- `numero_valido( quantità )` : determina se la quantità richiesta dell'articolo è un numero valido dentro i *range* prestabiliti:

Articoli generali	Articoli ristretti
`quantità > 0`	`quantità ≥ 0 AND quantità ≤ max_ammesso`

- `può_essere_soddisfatta( quantità )` : testa se la quantità richiesta può essere soddisfatta con lo stock disponibile:

Articoli generali	Articoli ristretti
`Stock_fisico + stock_in_transito - backorders ≥ quantità`	`stock_fisico ≥ quantità`

- `ritirare( quantità )`: riduce lo stock disponibile e incrementa i backorder, se corrisponde:

Articoli generali	Articoli ristretti

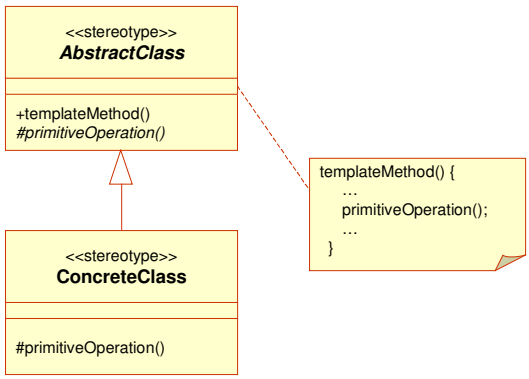
`stock_fisico=stock_fisico-quantità;`
`if ( stock_fisico < 0 )`
`  backorders = ABS( onHand );`
`  stock_fisico = 0;`
`endif`
	`stock_fisico = stock_fisico - quantità;`

Il problema consiste nella riutilizzazione della logica comune di un processo, nei confronti di classi diverse.

22.3. Descrizione della soluzione offerta dal pattern

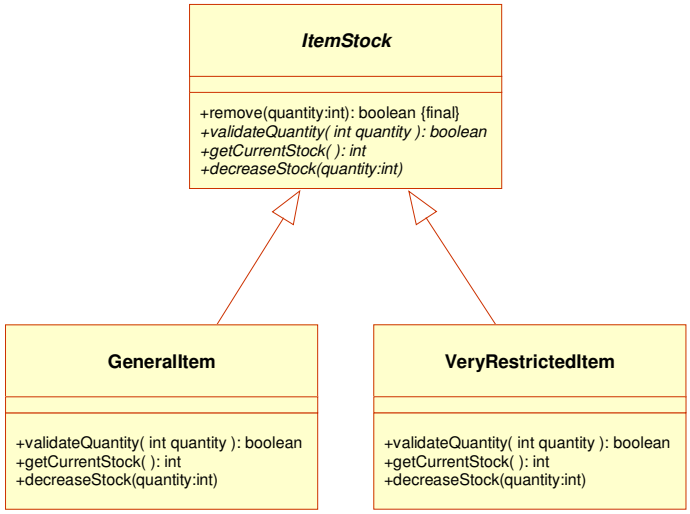
Il "*Template Method*" presenta una classe (AbstractClass) che implementa la logica del processo da riutilizzare, in funzione di certe operazioni (PrimitiveOperation) dichiarate in questa classe, la cui implementazione è di responsabilità delle sottoclassi (ConcreteClass).

22.4. Struttura del Pattern



22.5. Applicazione del Pattern

Schema del modello



Partecipanti

- **AbstractClass**: classe ItemStock.
  - Specifica le operazioni primitive astratte che le concrete sottoclassi devono definire per implementare i passi dell'algoritmo.
  - Implementa il template method creando la struttura di un algoritmo. Il template method invoca le operazioni primitive dichiarate nell'AbstractClass oppure quelle di altri oggetti.
- **ConcreteClass**: classi GeneralItem e VeryRestrictedItem.

- Implementano le operazioni primitive per eseguire i passi specifici dell'algoritmo.

Implementazione

Si presenta per primo la classe astratta `ItemStock`, che fornisce la logica generale del metodo `remove`. Le operazioni primitive invocate da questo metodo sono dichiarati come metodi astratti (`validateQuantity`, `getCurrentStock` e `decreaseStock`) :

```
public abstract class ItemStock {

  public final boolean remove(int quantity) {

     if( !validateQuantity( quantity ) )
        return false;
     if( getCurrentStock() >= quantity ) {
       decreaseStock( quantity );
       return true;
     }
     return false;
  }

  public abstract boolean validateQuantity( int quantity );
  public abstract int getCurrentStock( );
  public abstract void decreaseStock( int quantity );

}
```

La classe `GeneralItem` fornisce il codice dell'articolo generale. Oltre al nome e codice dell'articolo, questa classe possiede tre attributi: `onHand` (stock fisico) `inTransit` (stock in transito) e `backorders`. Si noti che s'implementano i metodi `getCurrentStock`, `decreaseStock` e `validateQuantity`:

```
public class GeneralItem extends ItemStock{

    String code;
    String name;
    int onHand;
    int inTransit;
    int backorders;

    public GeneralItem ( String cod, String nam ) {
        code = cod;
        name = nam;
    }

    public void setStock ( int oh, int it, int bo ) {
        onHand     = oh;
        inTransit  = it;
        backorders = bo;
    }

    public String toString() {
        return code + " " + name + ". Inv. On Hand: " +  onHand +
             " In Transit: " + inTransit + " Backorders: " + backorders;
    }

    public boolean validateQuantity( int quantity ) {
        return ( quantity >= 0 );
    }

    public int getCurrentStock() {
        return onHand + inTransit - backorders;
    }

    public void decreaseStock( int quantity ) {
        onHand -= quantity;
```

```
        if ( onHand < 0 ) {
           backorders += Math.abs( onHand );
           onHand = 0;
        }
    }

}
```

L'implementazione del `VeryRestrictedItem` utilizza la variabile `currentQuantity` per rappresentare lo stock fisico e la variabile `maxLotSize` per la massima quantità a prelevare in ogni occasione.

```
public class VeryRestrictedItem extends ItemStock{

    String code;
    String name;
    int currentQuantity;
    int maxLotSize;

    public VeryRestrictedItem ( String cod, String nam, int mlSize ) {
        code = cod;
        name = nam;
        maxLotSize = mlSize;
    }

    public void setStock ( int quantity ) {
        currentQuantity = quantity;
    }

    public String toString() {
        return code + " " + name + " " +  currentQuantity +
               " (Max. Lot Size: " + maxLotSize+ ")";
    }

    public int getCurrentStock() {
        return currentQuantity;
    }

    public void decreaseStock( int quantity ) {
        currentQuantity -= quantity;
    }

    public boolean validateQuantity( int quantity ) {
     if( quantity >= 0 &&
         quantity <= Math.min(currentQuantity, maxLotSize ) )
       return true;
     return false;
  }

}
```

Il codice che si presenta a continuazione, istanza prima un oggetto della classe `VeryRestrictedItem` e prova a fare richieste di consegna di alcune unità. Le stesse operazioni poi vengono fatte con un oggetto della classe `GeneralItem`:

```
public class TemplateMethodExample {

    public static void main( String[] arg ) {

      System.out.println( "A very restricted item example" );
      VeryRestrictedItem vri =
                  new VeryRestrictedItem( "VRI1" , "Golden stone", 5 );
      vri.setStock( 20 );
      System.out.println( "Current stock " + vri.toString() );
      System.out.println( "I will try to take 6 units:" );
      if( vri.remove( 6 ) )
        System.out.println( "Items removed." );
      else
        System.out.println( "Items cannot be removed." );
      System.out.println( vri.toString() );
      System.out.println( "I will try to take 3 units:" );
      if( vri.remove( 3 ) )
```

```
        System.out.println( "Items removed." );
      else
        System.out.println( "Items cannot be removed." );
      System.out.println( vri.toString() );

      System.out.println( "A general item example" );
      GeneralItem gi = new GeneralItem( "GEN1" , "Common stone" );
      gi.setStock( 20, 10, 0 );
      System.out.println( "Current stock " + gi.toString() );
      System.out.println( "I will try to take 6 units:" );
      if( gi.remove( 6 ) )
        System.out.println( "Items removed." );
      else
        System.out.println( "Items cannot be removed." );
      System.out.println( gi.toString() );
      System.out.println( "I will try to take 17 units:" );
      if( gi.remove( 17 ) )
        System.out.println( "Items removed." );
      else
        System.out.println( "Items cannot be removed." );
      System.out.println( gi.toString() );

    }
}
```

Osservazioni sull'esempio

Si noti che i due tipi d'oggetti rappresentano il loro stato interno tramite variabili diverse, ma in entrambi casi la logica del metodo di ritiro è stata riutilizzata.

Esecuzione dell'esempio

```
C:\Design Patterns\Behavioral\Temp method>java TemplateMethodExample

A very restricted item example

Current stock VRI1 Golden stone 20 (Max. Lot Size: 5)

I will try to take 6 units:
Items cannot be removed.
VRI1 Golden stone 20 (Max. Lot Size: 5)

I will try to take 3 units:
Items removed.
VRI1 Golden stone 17 (Max. Lot Size: 5)

A general item example

Current stock GEN1 Common stone. Inv. On Hand: 20 In Transit: 10
Backorders: 0

I will try to take 6 units:
Items removed.
GEN1 Common stone. Inv. On Hand: 14 In Transit: 10 Backorders: 0

I will try to take 17 units:
Items removed.
GEN1 Common stone. Inv. On Hand: 0 In Transit: 10 Backorders: 3
```

22.6. Osservazioni sull'implementazione in Java

La dichiarazione del template method come final, consente di rendere immodificabile la sua logica generale.

23. Visitor

(GoF pag. 331)

23.1. 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.

23.2. 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:

```
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() );
   }
}
```

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`.

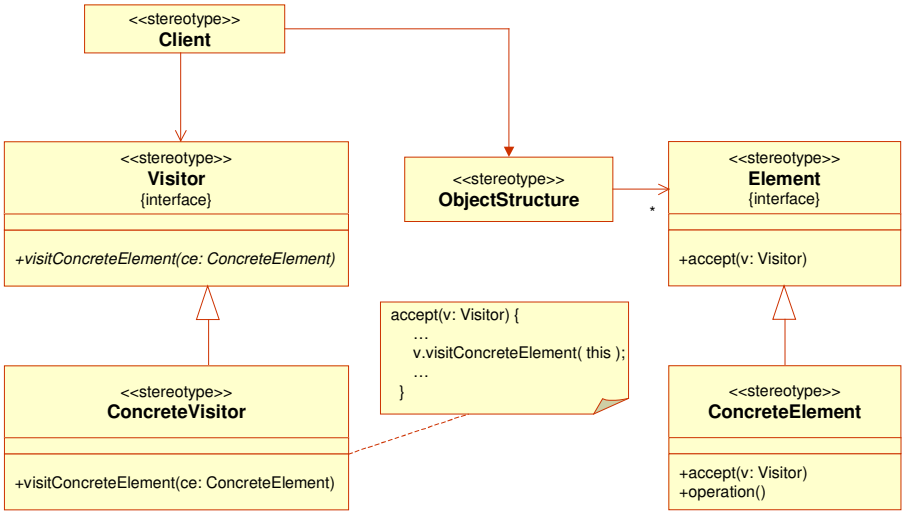
Il problema è trovare un modo di applicare questa operazione a tutti gli oggetti, senza includerla nel codice delle classi degli oggetti.

23.3. 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

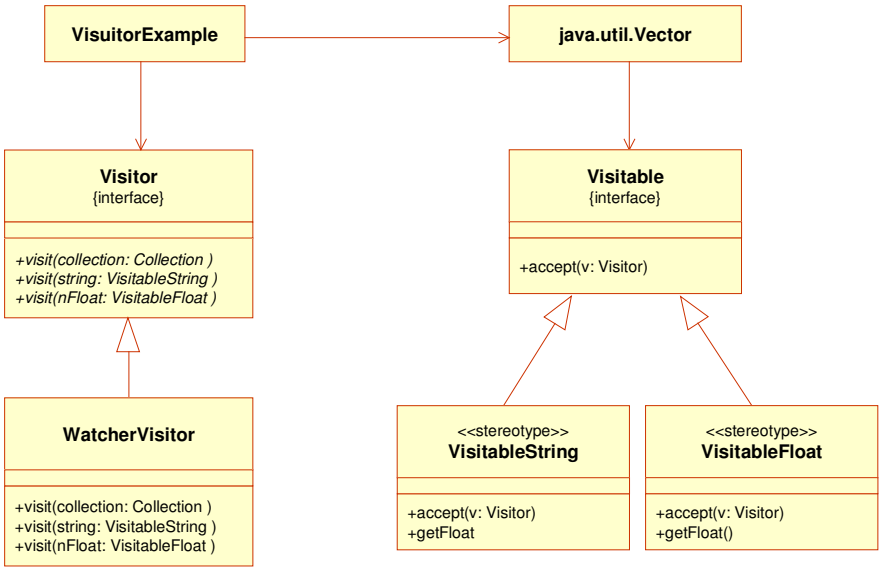
oggetto ricevuto, fa l'invocazione alla particolare operazione che in ogni caso si deve eseguire.

23.4. Struttura del Pattern



23.5. Applicazione del Pattern

Schema del modello



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**.
- **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 **ConcreteVisitor**:

```
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.

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

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

23.6. 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 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[13].
- 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()) {
         Object o = iterator.next();
         visit( o );
      }
   }

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

   public void visit(Float vFloat) {
```

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

```
      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`"; e
- 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 );

   }

}
```

Esecuzione dell'esempio

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

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

Riferimenti

[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 tuto*rial. 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 (visitata il 24/04/2001)

[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.