Indice

|  |
| --- |
|  |

12 – Design pattern 2

12.01 – Singleton pattern 5

12.02 – Observer pattern 7

12.03 – Template pattern 11

12.04 – Factory pattern 14

12.05 – Abstract Factory pattern 17

12.06 – Proxy pattern 24

12.07 – Iterator pattern 27

12.08 – Facade pattern 30

12.09 – Decorator pattern 33

12.10 – Data Access Object pattern 36

12 – Design pattern

|  |
| --- |
|  |

I **design pattern** derivano dalle idee proposte da Christopher Alexander, che suggerì l’**esistenza** **di** alcuni **schemi di progettazione comuni**, che erano **intrinsecamente gradevoli ed efficaci**.

Per **schema** si intende la **descrizione di un problema** e l’**essenza** della **sua soluzione**, in modo che la soluzione possa essere riutilizzata in diverse impostazioni.

Il design pattern si potrebbe definire come una descrizione delle conoscenze e delle esperienze accumulate, una soluzione sicura a un problema comune.

Esistono **tre tipi** di design pattern:

1. **Strutturali**, sono pattern che risolvono, a livello di struttura di classi, problemi ricorrenti. Semplificano la gestione di strutture complesse di classi;
2. **Creazionali**, sono pattern che risolvono, a livello di oggetti, problemi ricorrenti. Semplificano le istanziazioni di oggetti;
3. **Comportamentali**, sono pattern che consentono di definire comportamenti tra oggetti di classi diverse.

Gli **schemi** sono un modo di riutilizzare le conoscenze e le esperienze di altri progettisti; di solito sono associati alla progettazione orientata agli oggetti.

La [Gang of Four](https://it.wikipedia.org/wiki/Design_Patterns#:~:text=Nel%20libro%20gli%20autori%2C%20detti%20Gang%20of%20Four%2C,dando%20cos%C3%AC%20origine%20al%20cosiddetto%20linguaggio%20di%20pattern.) ha definito i **quattro elementi fondamentali degli schemi di progettazione**:

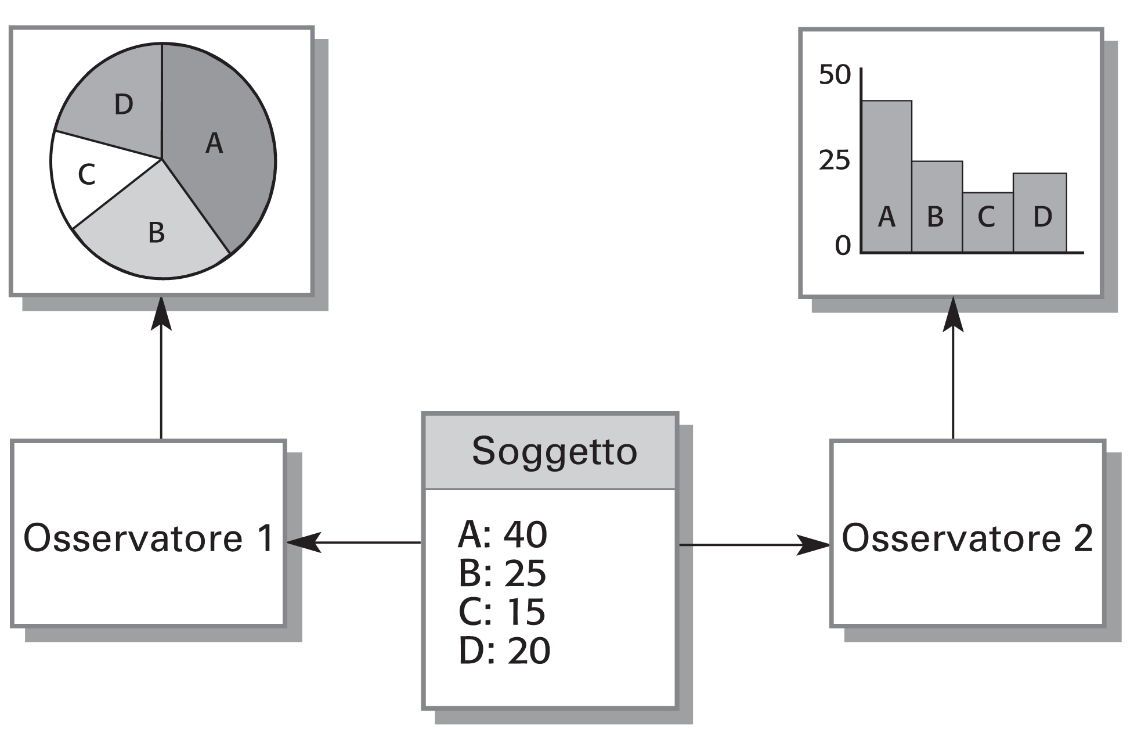
1. **Nome significativo** per riferirsi allo schema;
2. Una **descrizione dell’area del problema** per spiegare quando lo schema può essere applicato;
3. Una **descrizione delle parti della soluzione** del progetto, delle loro relazioni e responsabilità. Non è una descrizione concreta del progetto; è un modello per una soluzione del progetto che può essere istanziata in diversi modi. Viene solitamente espressa graficamente e mostra le relazioni tra gli oggetti e le classi degli oggetti;
4. Una **dichiarazione delle conseguenze** (risultati e compromessi) **dell’applicazione dello schema**. Questo può aiutare i progettisti a capire se uno schema può essere applicato efficacemente a una particolare situazione.

Gamma e i suoi coautori hanno suddiviso la descrizione di un problema in motivazione (una descrizione del perché lo schema è utile) e applicabilità (una descrizione dei casi in cui può essere applicato lo schema). La descrizione della soluzione include la struttura dello schema, i partecipanti, le collaborazioni e l’implementazione.

In figura viene presentato lo schema descrittivo e le successive presentazioni grafiche dello stesso insieme di dati.

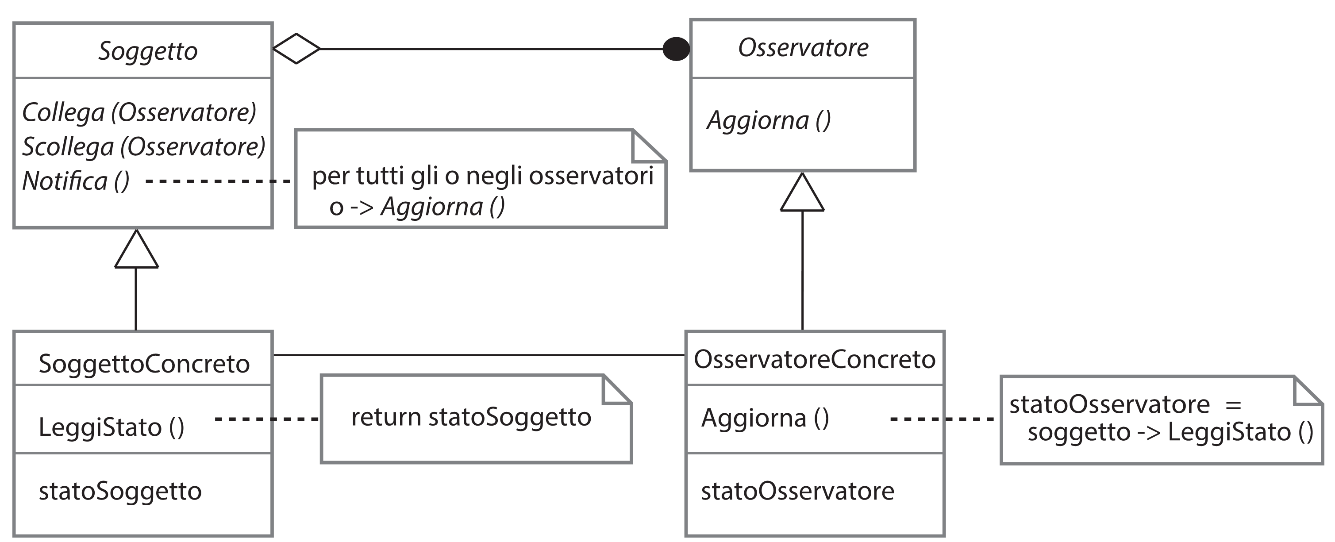
Immagine che contiene testo

Descrizione generata automaticamente



Le **rappresentazioni grafiche** sono **utilizzate** **per illustrare le classi di oggetti negli schemi e le loro relazioni**. Queste rappresentazioni **sono un supplemento alla descrizione degli schemi** e **aggiungono nuovi dettagli alla descrizione della soluzione**.

La seguente figura è la rappresentazione dello schema Observer nel linguaggio UML.



Gli schemi di progettazione sono una grande idea, ma occorre esperienza di progettazione del software per poterli utilizzare efficacemente. Occorre saper riconoscere i casi in cui uno schema può essere applicato.

|  |
| --- |
|  |

12.01 – Singleton pattern

Il singleton pattern, di tipo **creazionale**, è il miglior modo per creare un oggetto.

Questo pattern invoca una sola classe, la quale è responsabile della creazione di un solo e unico oggetto. Inoltre, la classe mette a disposizione un metodo per accedere al suo unico oggetto, al quale può essere fatto l’accesso senza necessità di istanziare l’oggetto della classe (si vedano gli esempi successivi per comprendere meglio).

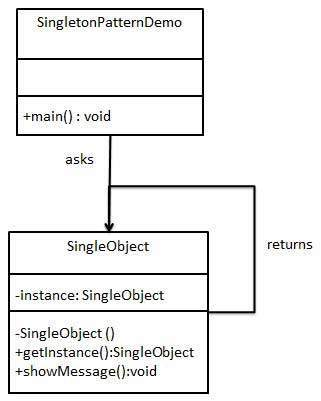
**Implementazione**

L’implementazione prevedere la creazione di una classe chiamata “SingleObject”. Essa ha il suo **costruttore privato** e ha una istanza di sé stesso di tipo “static”.

La classe “SingleObject” fornisce un metodo statico per ottenere la sua istanza statica all’esterno della classe.

La classe “SingletonPatternDemo” è una classe demo che utilizza la classe “SingleObject” per ottenre un oggetto “SingleObject”.

Il diagramma delle classi:



**Implementazione effettiva**

**Passo 1**

Viene creata una classe di tipo “Singleton”. Quindi, il file *SingleObject.java*:

**public class** SingleObject {  
  
 *//create an object of SingleObject* **private static** SingleObject *instance* = **new** SingleObject();  
  
 *//make the constructor private so that this class cannot be  
 //instantiated* **private** SingleObject(){}  
  
 *//Get the only object available* **public static** SingleObject getInstance(){  
 **return** *instance*;  
 }  
  
 **public void** showMessage(){  
 System.***out***.println(**"Hello World!"**);  
 }  
}

**Passo 2**

Dalla classe demo (*SingletonPatternDemo.java*), si ottiene solamente l’oggetto della classe singleton.

**public class** SingletonPatternDemo {  
 **public static void** main(String[] args) {  
  
 *//illegal construct  
 //Compile Time Error: The constructor SingleObject() is not visible  
 //SingleObject object = new SingleObject();  
  
 //Get the only object available* SingleObject object = SingleObject.*getInstance*();  
  
 *//show the message* object.showMessage();  
 }  
}

**Output**

Hello World!

|  |
| --- |
|  |

12.02 – Observer pattern

Il pattern “Observer” viene utilizzato quando c’è una relazione di tipo “uno-a molti” tra gli oggetti, per esempio se un oggetto viene modificato, i suoi oggetti dipendenti vengono modificati automaticamente.

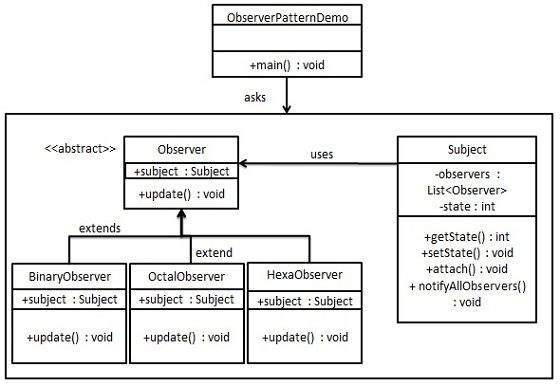
Questo pattern rientra nella categoria **comportamentale**.

**Implementazione**

Il pattern “Observer” utilizza tre classi di tipo “attore”: *Subject*, *Observer* e *Client*. La classe *Subject* è un oggetto avente metodi che consentono di attaccare e staccare *observer* da un oggetto di tipo *client*. Si crea una classe astratta (*abstract*) chiamata *Observer* e una classe concreta chiamata *Subject* che è l’estensione della classe *Observer*.

La classe demo *ObserverPatternDemo* utilizzerà *Subject* e l’oggetto della classe per mostrare il pattern in questione in azione.

Il diagramma delle classi:



**Implementazione effettiva**

**Passo 1**

Viene creata la classe *Subcject*:

**import** java.util.ArrayList;  
**import** java.util.List;  
  
**public class** Subject {  
  
 **private** List<Observer> **observers** = **new** ArrayList<Observer>();  
 **private int state**;  
  
 **public int** getState() {  
 **return state**;  
 }  
  
 **public void** setState(**int** state) {  
 **this**.**state** = state;  
 notifyAllObservers();  
 }  
  
 **public void** attach(Observer observer) {  
 **observers**.add(observer);  
 }  
  
 **public void** notifyAllObservers() {  
 **for** (Observer observer : **observers**) {  
 observer.update();  
 }  
 }  
}

**Passo 2**

Viene creata la classe *Observer*:

**public abstract class** Observer {  
 **protected** Subject **subject**;  
 **public abstract void** update();  
}

**Passo 3**

Vengono create le classi concrete *BinaryObserver*, *OctalObserver*, *HexaObserver*:

**public class** BinaryObserver **extends** Observer{  
  
 **public** BinaryObserver(Subject subject){  
 **this**.**subject** = subject;  
 **this**.**subject**.attach(**this**);  
 }  
  
 @Override  
 **public void** update() {  
 System.***out***.println(**"Binary String: "** + Integer.*toBinaryString*(**subject**.getState()));  
 }  
}

**public class** OctalObserver **extends** Observer{  
  
 **public** OctalObserver(Subject subject){  
 **this**.**subject** = subject;  
 **this**.**subject**.attach(**this**);  
 }  
 @Override  
 **public void** update() {  
 System.***out***.println(**"Octal String: "** + Integer.*toOctalString*(**subject**.getState()));  
 }  
}

**public class** HexaObserver **extends** Observer{  
  
 **public** HexaObserver(Subject subject){  
 **this**.**subject** = subject;  
 **this**.**subject**.attach(**this**);  
 }  
  
 @Override  
 **public void** update() {  
 System.***out***.println(**"Hex String: "** +  
 Integer.*toHexString*(**subject**.getState()).toUpperCase());  
 }  
}

**Passo 4**

Si utilizza *Subject* e oggetti *observer*:

**public class** ObserverPatternDemo {  
 **public static void** main(String[] args) {  
 Subject subject = **new** Subject();  
  
 **new** HexaObserver(subject);  
 **new** OctalObserver(subject);  
 **new** BinaryObserver(subject);  
  
 System.***out***.println(**"First state change: 15"**);  
 subject.setState(15);  
 System.***out***.println(**"Second state change: 10"**);  
 subject.setState(10);  
 }  
}

**Output**

First state change: 15

Hex String: F

Octal String: 17

Binary String: 1111

Second state change: 10

Hex String: A

Octal String: 12

Binary String: 1010

|  |
| --- |
|  |

12.03 – Template pattern

Nel pattern “template”, una classe astratta espone modi/templates definiti per eseguire i suoi metodi. Le sue sottoclassi possono eseguire l’*override*, ma l’invocazione deve essere effettuata nello stesso modo in cui viene fatto nella classe astratta.

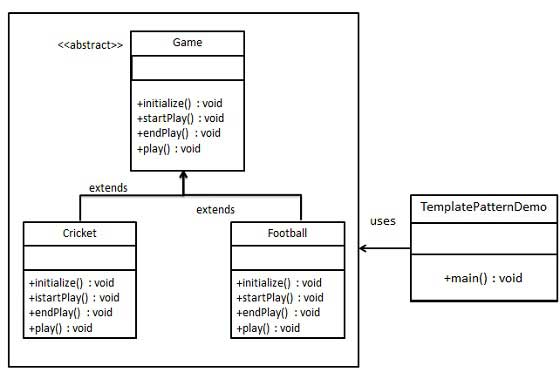
Questo pattern appartiene alla categoria **comportamentale**.

**Implementazione**

Si crea una classe astratta *Game* definendo le operazioni con un metodo impostato a “final” così da evitare l’*override*. Le classi *Cricket* e *Football* estendono *Game* ed eseguono l’*override* dei suoi metodi.

La classe demo *TemplatePatternDemo* utilizzerà la classe *Game* per dimostrare questo pattern.

Il diagramma delle classi:



**Implementazione effettiva**

**Passo 1**

Creazione di una classe astratta con un metodo template definito a “final”:

**public abstract class** Game {  
 **abstract void** initialize();  
 **abstract void** startPlay();  
 **abstract void** endPlay();  
  
 *//template method* **public final void** play() {  
 *//initialize the game* initialize();  
  
 *//start game* startPlay();  
  
 *//end game* endPlay();  
 }  
}

**Passo 2**

Creazione di classi che estendono *Game*:

**public class** Cricket **extends** Game{  
 @Override  
 **void** endPlay() {  
 System.***out***.println(**"Cricket Game Finished!"**);  
 }  
 @Override  
 **void** initialize() {  
 System.***out***.println(**"Cricket Game Initialized! Start playing."**);  
 }  
 @Override  
 **void** startPlay() {  
 System.***out***.println(**"Cricket Game Started. Enjoy the game!"**);  
 }  
}

**public class** Football **extends** Game{  
  
 @Override  
 **void** endPlay() {  
 System.***out***.println(**"Football Game Finished!"**);  
 }  
  
 @Override  
 **void** initialize() {  
 System.***out***.println(**"Football Game Initialized! Start playing."**);  
 }  
  
 @Override  
 **void** startPlay() {  
 System.***out***.println(**"Football Game Started. Enjoy the game!"**);  
 }  
}

**Passo 3**

Utilizzo del template *play* di *Game* per dimostrare un modo di giocare:

**public class** TemplatePatternDemo {  
 **public static void** main(String[] args) {  
  
 Game game = **new** Cricket();  
 game.play();  
 System.***out***.println();  
 game = **new** Football();  
 game.play();  
 }  
}

**Output**

Cricket Game Initialized! Start playing.

Cricket Game Started. Enjoy the game!

Cricket Game Finished!

Football Game Initialized! Start playing.

Football Game Started. Enjoy the game!

Football Game Finished!

|  |
| --- |
|  |

12.04 – Factory pattern

Il factory pattern è uno dei design pattern più utilizzati in Java. Questo tipo risiede nella categoria **creazionale** poiché fornisce la metodologia migliore per creare un oggetto.

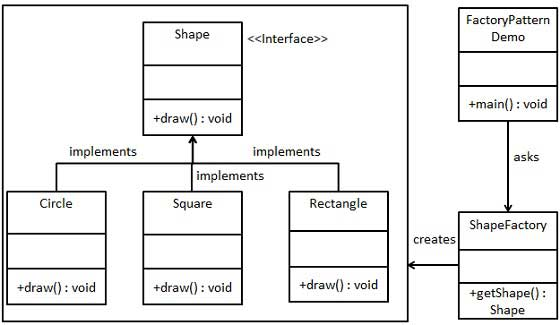
Nel factory pattern, si crea un oggetto senza esporre la logica della creazione al *client* e ci si riferisce ad esso utilizzando un’interfaccia comune.

**Implementazione**

Viene creata un’interfaccia chiamata *Shape* e le altre classi implementano l’interfaccia. Una classe factory chiamata *ShapeFactory* è definita di seguito.

La classe demo *FactoryPatternDemo* utilizza *ShapeFactory* per ottenere un oggetto *Shape*. Vengono passate informazioni (*CIRCLE* / *RECTANGLE* / *SQUARE*) alla classe *ShapeFactory* per ottenere il tipo necessario.

Il diagramma delle classi:



**Implementazione effettiva**

**Passo 1**

Creazione di un’interfaccia *Shape*:

**public interface** Shape {  
 **void** draw();  
}

**Passo 2**

Creazione di classi che implementino l’interfaccia:

**public class** Rectangle **implements** Shape{  
  
 @Override  
 **public void** draw() {  
 System.***out***.println(**"Inside Rectangle::draw() method."**);  
 }  
}

**public class** Square **implements** Shape{  
  
 @Override  
 **public void** draw() {  
 System.***out***.println(**"Inside Square::draw() method."**);  
 }  
}

**public class** Circle **implements** Shape{  
  
 @Override  
 **public void** draw() {  
 System.***out***.println(**"Inside Circle::draw() method."**);  
 }  
}

**Passo 3**

Creazione di una *Factory* per generare oggetti, per le classi, basati sulle informazioni fornite:

**public class** ShapeFactory {  
  
 *//use getShape method to get object of type shape* **public** Shape getShape(String shapeType){  
 **if**(shapeType == **null**){  
 **return null**;  
 }  
 **if**(shapeType.equalsIgnoreCase(**"CIRCLE"**)){  
 **return new** Circle();  
 } **else if**(shapeType.equalsIgnoreCase(**"RECTANGLE"**)){  
 **return new** Rectangle();  
 } **else if**(shapeType.equalsIgnoreCase(**"SQUARE"**)){  
 **return new** Square();  
 }  
  
 **return null**;  
 }  
}

**Passo 4**

Utilizzo della *Factory* per ottenere l’oggetto di una classe passando un’informazione come il tipo:

**public class** FactoryPatternDemo {  
  
 **public static void** main(String[] args) {  
 ShapeFactory shapeFactory = **new** ShapeFactory();  
  
 *//get an object of Circle and call its draw method.* Shape shape1 = shapeFactory.getShape(**"CIRCLE"**);  
  
 *//call draw method of Circle* shape1.draw();  
  
 *//get an object of Rectangle and call its draw method.* Shape shape2 = shapeFactory.getShape(**"RECTANGLE"**);  
  
 *//call draw method of Rectangle* shape2.draw();  
  
 *//get an object of Square and call its draw method.* Shape shape3 = shapeFactory.getShape(**"SQUARE"**);  
  
 *//call draw method of circle* shape3.draw();  
 }  
}

**Output**

Inside Circle::draw() method.

Inside Rectangle::draw() method.

Inside Square::draw() method.

|  |
| --- |
|  |

12.05 – Abstract Factory pattern

L’abstract factory pattern lavora intorno una super-factory la quale crea altre factories. Quest’ultima è anche chiamata factory di factories. Questo tipo di design pattern è nella categoria **creazionale** e fornisce uno dei modi **migliori** per creare un oggetto.

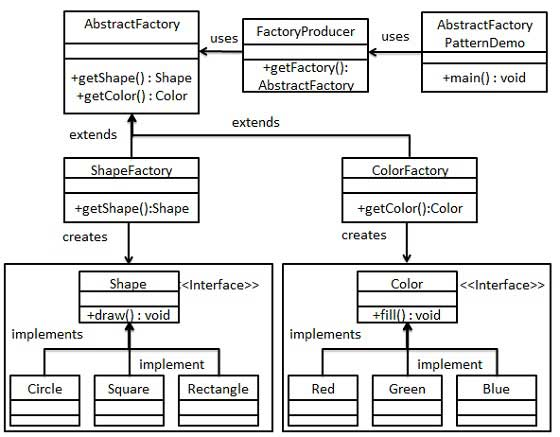
Inoltre, una interfaccia è responsabile di una creazione di una factory di oggetti correlati senza esplicitare specificatamente le loro classi. Ogni factory generata può dare gli oggetti secondo il factory pattern.

**Implementazione**

Vengono create due interfacce *Shape* e *Color* e delle classi che implementano queste interfacce. Quindi, si crea una classe abstract factory *AbstractFactory*. Le classi factory *ShapeFactory* e *ColorFactory* sono definite ed estendono *AbstractFactory*. Una classe factory generatrice/creatrice *FactoryProducer* viene creata.

La classe demo *AbstractFactoryPatternDemo* utilizza *FactoryProducer* per ottenere un oggetto *AbstractFactory*. Vengono passate informazioni (*CIRCLE* / *RECTANGLE* / *SQUARE*) alla classe *AbstractShapeFactory* per ottenere il tipo necessario. Inoltre, vengono passate informazioni (*RED* / *GREEN* / *BLUE* per *Color*) alla classe *AbstractFactory* per ottenere il tipo d’oggetto richiesto.

Il diagramma delle classi:



**Implementazione effettiva**

**Passo 1**

Creazione di un’interfaccia *Shape*:

**public interface** Shape {  
 **void** draw();  
}

**Passo 2**

Creazione di una classe implementando la stessa interfaccia:

**import** factory.Shape;  
  
**public class** Rectangle **implements** Shape {  
  
 @Override  
 **public void** draw() {  
 System.***out***.println(**"Inside Rectangle::draw() method."**);  
 }  
}

**import** factory.Shape;  
  
**public class** Square **implements** Shape {  
  
 @Override  
 **public void** draw() {  
 System.***out***.println(**"Inside Square::draw() method."**);  
 }  
}

**import** factory.Shape;  
  
**public class** Circle **implements** Shape {  
  
 @Override  
 **public void** draw() {  
 System.***out***.println(**"Inside Circle::draw() method."**);  
 }  
}

**Passo 3**

Creazione di una interfaccia di colori:

**public interface** Color {  
 **void** fill();  
}

**Passo 4**

Creazione di una classe che implementa la stessa interfaccia:

**public class** Red **implements** Color{  
  
 @Override  
 **public void** fill() {  
 System.***out***.println(**"Inside Red::fill() method."**);  
 }  
}

**public class** Green **implements** Color{  
  
 @Override  
 **public void** fill() {  
 System.***out***.println(**"Inside Green::fill() method."**);  
 }  
}

**public class** Blue **implements** Color{  
  
 @Override  
 **public void** fill() {  
 System.***out***.println(**"Inside Blue::fill() method."**);  
 }  
}

**Passo 5**

Creazione di una classe astratta per ottenere “factories” degli oggetti *Color* e *Shape*:

**public abstract class** AbstractFactory {  
 **abstract** Color getColor(String color);  
 **abstract** Shape getShape(String shape);  
}

**Passo 6**

Creazione di classi Factory estendendo *AbstractFactory* per generare oggetti di classi basate sulle informazioni fornite:

**import** factory.Circle;  
**import** factory.Rectangle;  
**import** factory.Shape;  
**import** factory.Square;  
  
**public class** ShapeFactory **extends** AbstractFactory{  
  
 @Override  
 **public** Shape getShape(String shapeType){  
  
 **if**(shapeType == **null**){  
 **return null**;  
 }  
  
 **if**(shapeType.equalsIgnoreCase(**"CIRCLE"**)){  
 **return new** Circle();  
 } **else if**(shapeType.equalsIgnoreCase(**"RECTANGLE"**)){  
 **return new** Rectangle();  
 } **else if**(shapeType.equalsIgnoreCase(**"SQUARE"**)){  
 **return new** Square();  
 }  
  
 **return null**;  
 }  
  
 @Override  
 Color getColor(String color) {  
 **return null**;  
 }  
}

**public class** ColorFactory **extends** AbstractFactory{  
  
 @Override  
 **public** Shape getShape(String shapeType) {  
 **return null**;  
 }  
  
 @Override  
 Color getColor(String color) {  
  
 **if**(color == **null**){  
 **return null**;  
 }  
  
 **if**(color.equalsIgnoreCase(**"RED"**)){  
 **return new** Red();  
 }**else if** (color.equalsIgnoreCase(**"GREEN"**)) {  
 **return new** Green();  
 } **else if** (color.equalsIgnoreCase(**"BLUE"**)) {  
 **return new** Blue();  
 }  
  
 **return null**;  
 }  
}

**Passo 7**

Creazione di un generatore di classe Factory per ottenere le “factories” passando informazioni come la forma o il colore:

**public class** FactoryProducer {  
 **public static** AbstractFactory getFactory(String choice){  
  
 **if**(choice.equalsIgnoreCase(**"SHAPE"**)){  
 **return new** ShapeFactory();  
 } **else if** (choice.equalsIgnoreCase(**"COLOR"**)) {  
 **return new** ColorFactory();  
 }  
  
 **return null**;  
 }  
}

**Passo 8**

Utilizzo della *FactoryProducer* per ottenere *AbstractFactory* per ottenere “factories” di classi concrete passando informazioni come il tipo:

**public class** AbstractFactoryPatternDemo {  
 **public static void** main(String[] args) {  
  
 *//get shape factory* AbstractFactory shapeFactory = FactoryProducer.*getFactory*(**"SHAPE"**);  
  
 *//get an object of Shape Circle* Shape shape1 = shapeFactory.getShape(**"CIRCLE"**);  
  
 *//call draw method of Shape Circle* shape1.draw();  
  
 *//get an object of Shape Rectangle* Shape shape2 = shapeFactory.getShape(**"RECTANGLE"**);  
  
 *//call draw method of Shape Rectangle* shape2.draw();  
  
 *//get an object of Shape Square* Shape shape3 = shapeFactory.getShape(**"SQUARE"**);  
  
 *//call draw method of Shape Square* shape3.draw();  
  
 *//get color factory* AbstractFactory colorFactory = FactoryProducer.*getFactory*(**"COLOR"**);  
  
 *//get an object of Color Red* Color color1 = colorFactory.getColor(**"RED"**);  
  
 *//call fill method of Red* color1.fill();  
  
 *//get an object of Color Green* Color color2 = colorFactory.getColor(**"Green"**);  
  
 *//call fill method of Green* color2.fill();  
  
 *//get an object of Color Blue* Color color3 = colorFactory.getColor(**"BLUE"**);  
  
 *//call fill method of Color Blue* color3.fill();  
 }  
}

**Output**

Inside Circle::draw() method.

Inside Rectangle::draw() method.

Inside Square::draw() method.

Inside Red::fill() method.

Inside Green::fill() method.

Inside Blue::fill() method.

|  |
| --- |
|  |

12.06 – Proxy pattern

In un pattern proxy, una classe rappresenta le funzionalità di un’altra classe. Questo pattern rientra nella categoria dei pattern **strutturali**.

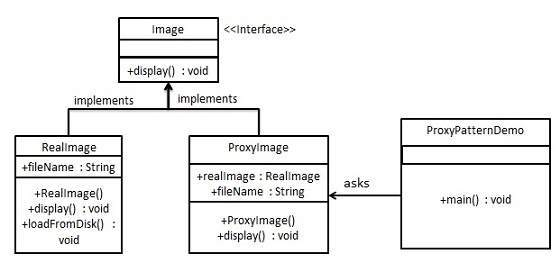
In un pattern proxy, vengono creati oggetti basati sull’oggetto originale, ma che consentono di interfacciarsi con il mondo esterno.

**Implementazione**

Viene creata una interfaccia *Image* e una classe che la implementa. La classe proxy *ProxyImage* riduce l’ingombro di memoria del caricamento di oggetti *RealImmage*.

La classe demo *ProxyPatternDemo* utilizza *ProxyImage* per ottenere un oggetto *Image* da caricare e mostrare a chi ne ha bisogno.

Il diagramma delle classi:



**Implementazione effettiva**

**Passo 1**

Creazione di un’interfaccia:

**public interface** Image {  
 **void** display();  
}

**Passo 2**

Creazione di una classe implementando l’interfaccia:

**public class** RealImage **implements** Image{  
  
 **private** String **fileName**;  
  
 **public** RealImage(String fileName){  
 **this**.**fileName** = fileName;  
 loadFromDisk(fileName);  
 }  
  
 @Override  
 **public void** display(){  
 System.***out***.println(**"Displaying "** + **fileName**);  
 }  
  
 **private void** loadFromDisk(String fileName){  
 System.***out***.println(**"Loading "** + fileName);  
 }  
}

**public class** ProxyImage **implements** Image{  
  
 **private** RealImage **realImage**;  
 **private** String **fileName**;  
  
 **public** ProxyImage(String fileName){  
 **this**.**fileName** = fileName;  
 }  
  
 @Override  
 **public void** display(){  
 **if**(**realImage** == **null**){  
 **realImage** = **new** RealImage(**fileName**);  
 }  
 **realImage**.display();  
 }  
}

**Passo 3**

Utilizzo della classe *ProxyImage* per ottenere l’oggetto della classe *RealImage* quando richiesto:

**public class** ProxyPatternDemo {  
  
 **public static void** main(String[] args) {  
 Image image = **new** ProxyImage(**"test\_10mb.jpg"**);  
  
 *//image will be loaded from disk* image.display();  
 System.***out***.println(**""**);  
  
 *//image will not be loaded from disk* image.display();  
 }  
}

**Output**

Loading test\_10mb.jpg

Displaying test\_10mb.jpg

Displaying test\_10mb.jpg

|  |
| --- |
|  |

12.07 – Iterator pattern

Il pattern iterator è molto utilizzato in java e in .Net. Lo scopo principale è quello di avere un accesso agli elementi di una collezione di oggetti in maniera sequenziale senza nessun bisogno di conoscere la sua sottostante rappresentazione.

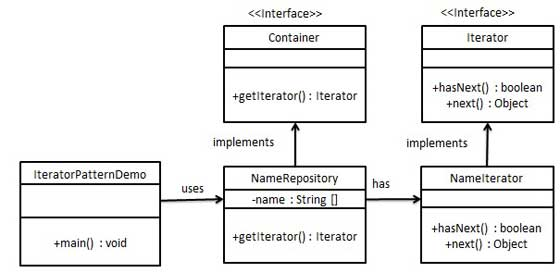
Il pattern iterator risiede nella categoria **comportamentale**.

**Implementazione**

Viene creata una interfaccia *Iterator* la quale descrive i metodi di navigazione e una interfaccia *Container* la quale ritorna l’iteratore. Le classi implementano l’interfaccia *Container* che sarà responsabile di implementare l’interfaccia *Iterator* e di utilizzarla.

La classe demo *IteratorPatternDemo* utilizzerà *NamesRepository*, una implementazione di una classe che stamperà un *Names* salvato come *collection* in *NamesRepository*.

Il diagramma delle classi:



**Implementazione effettiva**

**Passo 1**

Creazione di interfacce:

**public interface** Iterator {  
 **public boolean** hasNext();  
 **public** Object next();  
}

**public interface** Container {  
 **public** Iterator getIterator();  
}

**Passo 2**

Creazione di una classe che implementa l’interfaccia *Container*. Questa classe ha una classe interna *NameIterator* che implementa l’interfaccia *Iterator*:

**public class** NameRepository **implements** Container {  
 **public** String **names**[] = {**"Robert"**, **"John"**, **"Julie"**, **"Lora"**};  
  
 @Override  
 **public** Iterator getIterator() {  
 **return new** NameIterator();  
 }  
  
 **private class** NameIterator **implements** Iterator {  
  
 **int index**;  
  
 @Override  
 **public boolean** hasNext() {  
 **if**(**index** < **names**.**length**){  
 **return true**;  
 }  
 **return false**;  
 }  
   
 @Override  
 **public** Object next() {  
 **if**(**this**.hasNext()){  
 **return names**[**index**++];  
 }  
 **return null**;  
 }  
 }  
}

**Passo 3**

Utilizzo di *NameRepository* per ottenere l’iteratore e stampare i nomi:

**public class** IteratorPatternDemo {  
  
 **public static void** main(String[] args) {  
 NameRepository nameRepository = **new** NameRepository();  
  
 **for**(Iterator iter = nameRepository.getIterator(); iter.hasNext();){  
 String name = (String)iter.next();  
 System.***out***.println(**"Name : "** + name);  
 }  
 }  
}

**Output**

Name : Robert

Name : John

Name : Julie

Name : Lora

|  |
| --- |
|  |

12.08 – Facade pattern

Il pattern facade nasconde le complessità del sistema e fornisce un’interfaccia al *client*, il quale può accedere al sistema. Questo tipo di design pattern fa parte della categoria **strutturale** poiché aggiunge un’interfaccia ad un sistema esistente per nascondere le sue complessità.

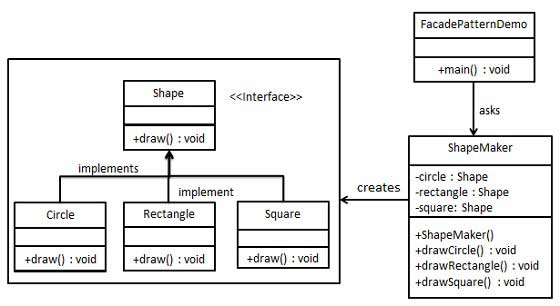
Questo pattern richiede una singola classe che si occuperà di semplificare i metodi richiesti dal *client* e delegare le chiamate ai metodi di classi di sistema esistenti.

**Implementazione**

Viene creata un’interfaccia *Shape* e una classe che la implementi. Una classe *facade ShapeMaker* è definita per delegare le chiamate degli utenti a queste classi.

La classe demo *FacadePatternDemo* utilizza la classe *ShapeMaker* per mostrare i risultati.

Il diagramma delle classi:



**Implementazione effettiva**

**Passo 1**

Creazione di un’interfaccia:

**public interface** Shape {  
 **void** draw();  
}

**Passo 2**

Creazione di una classe che implementi l’interfaccia precedente:

**public class** Rectangle **implements** Shape {  
  
 @Override  
 **public void** draw() {  
 System.***out***.println(**"Inside Rectangle::draw() method."**);  
 }  
}

**public class** Square **implements** Shape {  
  
 @Override  
 **public void** draw() {  
 System.***out***.println(**"Inside Square::draw() method."**);  
 }  
}

**public class** Circle **implements** Shape {  
  
 @Override  
 **public void** draw() {  
 System.***out***.println(**"Inside Circle::draw() method."**);  
 }  
}

**Passo 3**

Creazione di una classe facade:

**public class** ShapeMaker {  
 **private** Shape **circle**;  
 **private** Shape **rectangle**;  
 **private** Shape **square**;  
  
 **public** ShapeMaker() {  
 **circle** = **new** Circle();  
 **rectangle** = **new** Rectangle();  
 **square** = **new** Square();  
 }  
   
 **public void** drawCircle(){  
 **circle**.draw();  
 }  
 **public void** drawRectangle(){  
 **rectangle**.draw();  
 }  
 **public void** drawSquare(){  
 **square**.draw();  
 }  
}

**Passo 4**

Utilizzo della classe facade per disegnare vari tipi di forme:

**public class** FacadePatternDemo {  
 **public static void** main(String[] args){  
 ShapeMaker shapeMaker = **new** ShapeMaker();  
  
 shapeMaker.drawCircle();  
 shapeMaker.drawRectangle();  
 shapeMaker.drawSquare();  
 }  
}

**Output**

Inside Circle::draw() method.

Inside Rectangle::draw() method.

Inside Square::draw() method.

|  |
| --- |
|  |

12.09 – Decorator pattern

Il pattern decorator consente ad un utente di aggiungere nuove funzionalità ad un oggetto esistente senza alterare la sua struttura. Il decorator pattern rientra nella categoria **strutturale** poiché agisce come un *wrapper* di una classe esistente.

Questo pattern richiede una singola classe che si occuperà di semplificare i metodi richiesti dal *client* e delegare le chiamate ai metodi di classi di sistema esistenti.

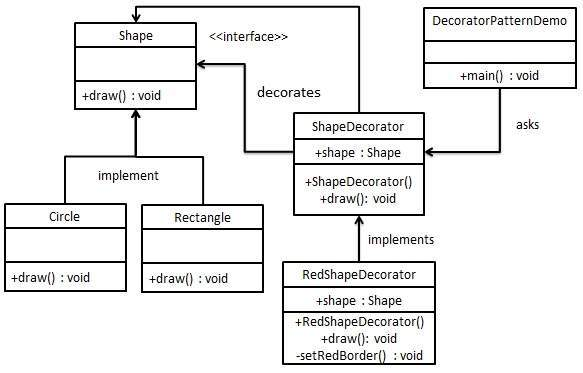
**Implementazione**

Viene creata un’interfaccia *Shape* e una classe che la implementi. Viene creata una classe astratta *decorator ShapeDecorator* che implementa l’interfaccia *Shape* e ha gli oggetti *Shape* come sue variabili.

*RedShapeDecorator* è una classe che implementa *ShapeDecorator*.

*DecoratorPatternDemo* è la classe demo che utilizza *RedShapeDecorator* per decorare gli oggetti *Shape*.

Il diagramma delle classi:



**Implementazione effettiva**

**Passo 1**

Creazione di un’interfaccia:

**public interface** Shape {  
 **void** draw();  
}

**Passo 2**

Creazione di classi che implementano la stessa interfaccia:

**public class** Rectangle **implements** Shape {  
  
 @Override  
 **public void** draw() {  
 System.***out***.println(**"Inside Rectangle::draw() method."**);  
 }  
}

**public class** Circle **implements** Shape {  
  
 @Override  
 **public void** draw() {  
 System.***out***.println(**"Inside Circle::draw() method."**);  
 }  
}

**Passo 3**

Creazione di una classe *decorator* astratta che implementa l’interfaccia *Shape*:

**public abstract class** ShapeDecorator **implements** Shape{  
 **protected** Shape **decoratedShape**;  
  
 **public** ShapeDecorator(Shape decoratedShape){  
 **this**.**decoratedShape** = decoratedShape;  
 }  
  
 **public void** draw(){  
 **decoratedShape**.draw();  
 }  
}

**Passo 4**

Creazione di una classe *decorator* che estende la classe *ShapeDecorator*:

**public class** RedShapeDecorator **extends** ShapeDecorator {  
  
 **public** RedShapeDecorator(Shape decoratedShape) {  
 **super**(decoratedShape);  
 }  
  
 @Override  
 **public void** draw() {  
 **decoratedShape**.draw();  
 setRedBorder(**decoratedShape**);  
 }  
  
 **private void** setRedBorder(Shape decoratedShape){  
 System.***out***.println(**"Border Color: Red"**);  
 }  
}

**Passo 5**

Utilizzo della *RedShapeDecorator* per decorare gli oggetti *Shape*:

**public class** DecoratorPatternDemo {  
 **public static void** main(String[] args) {  
  
 Shape circle = **new** Circle();  
  
 Shape redCircle = **new** RedShapeDecorator(**new** Circle());  
  
 Shape redRectangle = **new** RedShapeDecorator(**new** Rectangle());  
 System.***out***.println(**"Circle with normal border"**);  
 circle.draw();  
  
 System.***out***.println(**"\nCircle of red border"**);  
 redCircle.draw();  
  
 System.***out***.println(**"\nRectangle of red border"**);  
 redRectangle.draw();  
 }  
}

**Output**

Circle with normal border

Shape: Circle

Circle of red border

Shape: Circle

Border Color: Red

Rectangle of red border

Shape: Rectangle

Border Color: Red

|  |
| --- |
|  |

12.10 – Data Access Object pattern

Il pattern Data Access Object (DAO) viene utilizzato per separare l’accesso o operazioni su dati a basso livello dalla parte di alto livello. I seguenti partecipanti fanno parte di questo pattern:

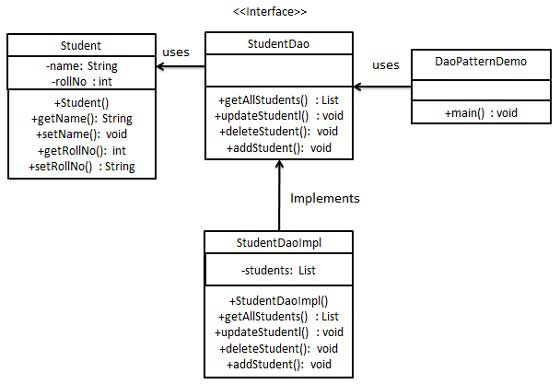
* **Data Access Object Interface**, questa interfaccia definisce le operazioni standard che possono essere eseguite in un oggetto/i.
* **Data Access Object concrete class**, questa classe implementa l’interfaccia precedente. Questa classe è responsabile di ottenere i dati da una risorsa che può essere un database, xml o un altro meccanismo di immagazzinamento.
* **Model Object or Value Object**, questo oggetto (POJO) contiene metodi per impostare o ottenere i dati recuperati usando una classe DAO.

**Implementazione**

Viene creato un oggetto *Student* rappresentandolo come un oggetto *Model* o *Value*. *StudentDao* è un *Data Access Object Interface*. *StudentDaoImpl* è una classe che implementa l’interfaccia precedente.

Infine, la classe demo *DaoPatternDemo* utilizza *StudentDao* per dimostrare l’uso del pattern *Data Access Object*.

Il diagramma delle classi:



**Implementazione effettiva**

**Passo 1**

Creazione di oggetto Value:

**public class** Student {  
 **private** String **name**;  
 **private int rollNo**;  
  
 Student(String name, **int** rollNo){  
 **this**.**name** = name;  
 **this**.**rollNo** = rollNo;  
 }  
  
 **public** String getName(){  
 **return name**;  
 }  
  
 **public void** setName(String name){  
 **this**.**name** = name;  
 }  
  
 **public int** getRollNo() {  
 **return rollNo**;  
 }  
  
 **public void** setRollNo(**int** rollNo) {  
 **this**.**rollNo** = rollNo;  
 }  
}

**Passo 2**

Creazione di un’interfaccia data access object:

**import** java.util.List;  
  
**public interface** StudentDao {  
 **public** List<Student> getAllStudents();  
 **public** Student getStudent(**int** rollNo);  
 **public void** updateStudent(Student student);  
 **public void** deleteStudent(Student student);  
}

**Passo 3**

Creazione di una classe che implementa l’interfaccia precedente:

**import** java.util.ArrayList;  
**import** java.util.List;  
  
**public class** StudentDaoImpl **implements** StudentDao {  
  
 *// list is working as a database* List<Student> **students**;  
  
 **public** StudentDaoImpl() {  
 **students** = **new** ArrayList<Student>();  
 Student student1 = **new** Student(**"Robert"**, 0);  
 Student student2 = **new** Student(**"John"**, 1);  
 **students**.add(student1);  
 **students**.add(student2);  
 }  
  
 @Override  
 **public void** deleteStudent(Student student) {  
 **students**.remove(student.getRollNo());  
 System.***out***.println(**"Student: Roll No "** + student.getRollNo() + **", deleted from db"**);  
 }  
  
 *//retrive list of students from the database* @Override  
 **public** List<Student> getAllStudents() {  
 **return students**;  
 }  
  
 @Override  
 **public** Student getStudent(**int** rollNo) {  
 **return students**.get(rollNo);  
 }  
  
 @Override  
 **public void** updateStudent(Student student) {  
 **students**.get(student.getRollNo()).setName(student.getName());  
 System.***out***.println(**"Student: Roll No "** + student.getRollNo() + **", updated in the db"**);  
 }  
}

**Passo 4**

Utilizzo del *StudentDao* per dimostrare l’utilizzo del pattern:

**public class** DaoPatternDemo {  
 **public static void** main(String[] args) {  
 StudentDao studentDao = **new** StudentDaoImpl();  
  
 *// print all students* **for** (Student student : studentDao.getAllStudents()) {  
 System.***out***.println(**"Student: [RollNo : "** + student.getRollNo() + **", Name : "** + student.getName() + **"]"**);  
 }  
  
  
 *// update student* Student student = studentDao.getAllStudents().get(0);  
 student.setName(**"Michael"**);  
 studentDao.updateStudent(student);  
  
 *// get the student* studentDao.getStudent(0);  
 System.***out***.println(**"Student : [RollNo : "** + student.getRollNo() + **", Name : "** + student.getName() + **"]"**);  
 }  
}

**Output**

Student: [RollNo : 0, Name : Robert]

Student: [RollNo : 1, Name : John]

Student: Roll No 0, updated in the db

Student : [RollNo : 0, Name : Michael]