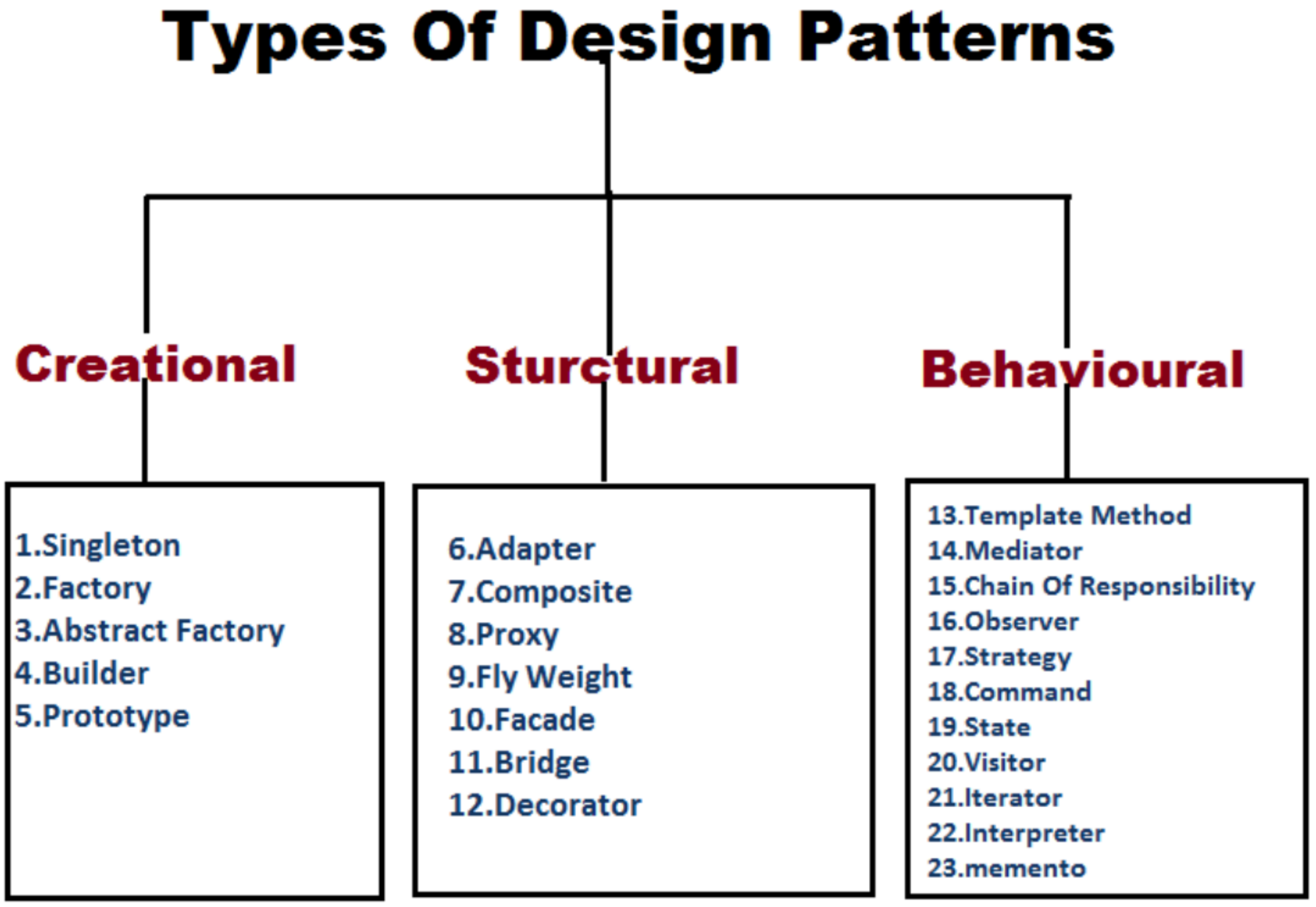
Design patterns

By Pakita Shamoi

Fall 2017

General idea

- “A pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.”
  - Patterns support reuse of software architecture and design
  - Automobile designers don't design cars from scratch using the laws of physics
  - Instead, they reuse standard designs with successful track records, learning from experience
- They are:
  - “Descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.”


Types Of Design Patterns
Creational
Sturctural
Behavioural
1.Singleton
2.Factory
3.Abstract Factory
4.Builder
5.Prototype
6.Adapter
7.Composite
8.Proxy
9.Fly Weight
10.Facade
11.Bridge
12.Decorator
13.Template Method
14.Mediator
15.Chain Of Responsibility
16.Observer
17.Strategy
18.Command
19.State
20.Visitor
21.Iterator
22.Interpreter
23.memento

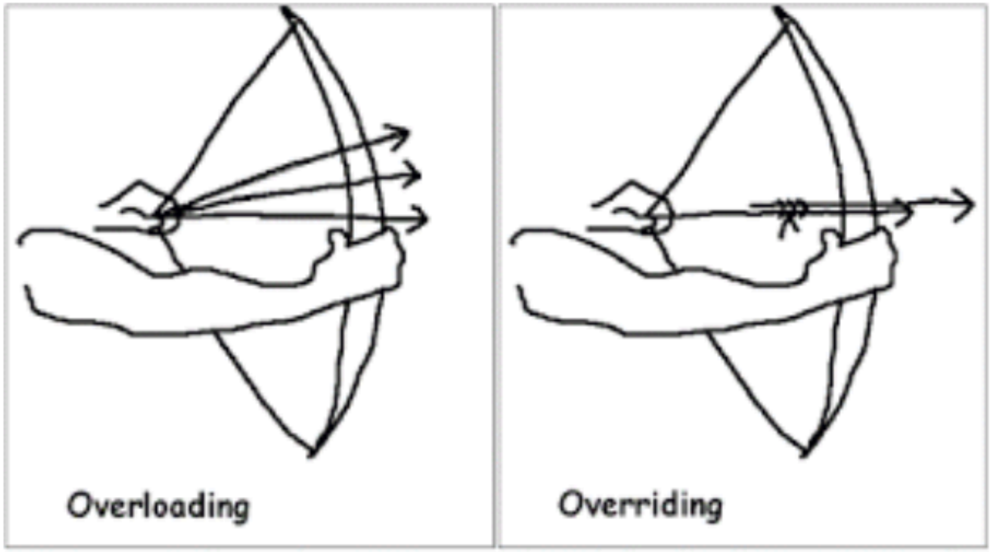
Types of patterns

- **Creational patterns:**
  - Deal with initializing and configuring classes and objects
- **Structural patterns**:
  - Deal with decoupling interface and implementation of classes and objects
  - Composition of classes or objects
- **Behavioral patterns**:
  - Deal with dynamic interactions among societies of classes and objects
  - How they distribute responsibility

Main OOP concepts

1. Abstraction
2. Encapsulation
3. Inheritance
4. Polymorphisam

Overloading and Overriding



Overloading is compile time Polymorphism and Overriding is runtime Polymorphism.

Is there any sense in having a constructor in the abstract class?

Identify which oops concept were used in below scenario:

”In a group of 5 boys one boy never give any contribution when group goes for eating out, party or anything.
Suddenly one beautiful girl joins the same group and then the aforementioned boy spends lots of money for the group.”

Is there any sense in a private constructor?

Singleton class

In software engineering, the singleton pattern is a software design pattern that restricts the instantiation of a class to one object. This is useful when exactly one object is needed to coordinate actions across the system.

```java
public final class Singleton {
    private static final Singleton INSTANCE = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return INSTANCE;
    }
}
```

Singleton class

An implementation of the singleton pattern must:

- Ensure that only one instance of the singleton class ever exists;
- and provide global access to that instance.

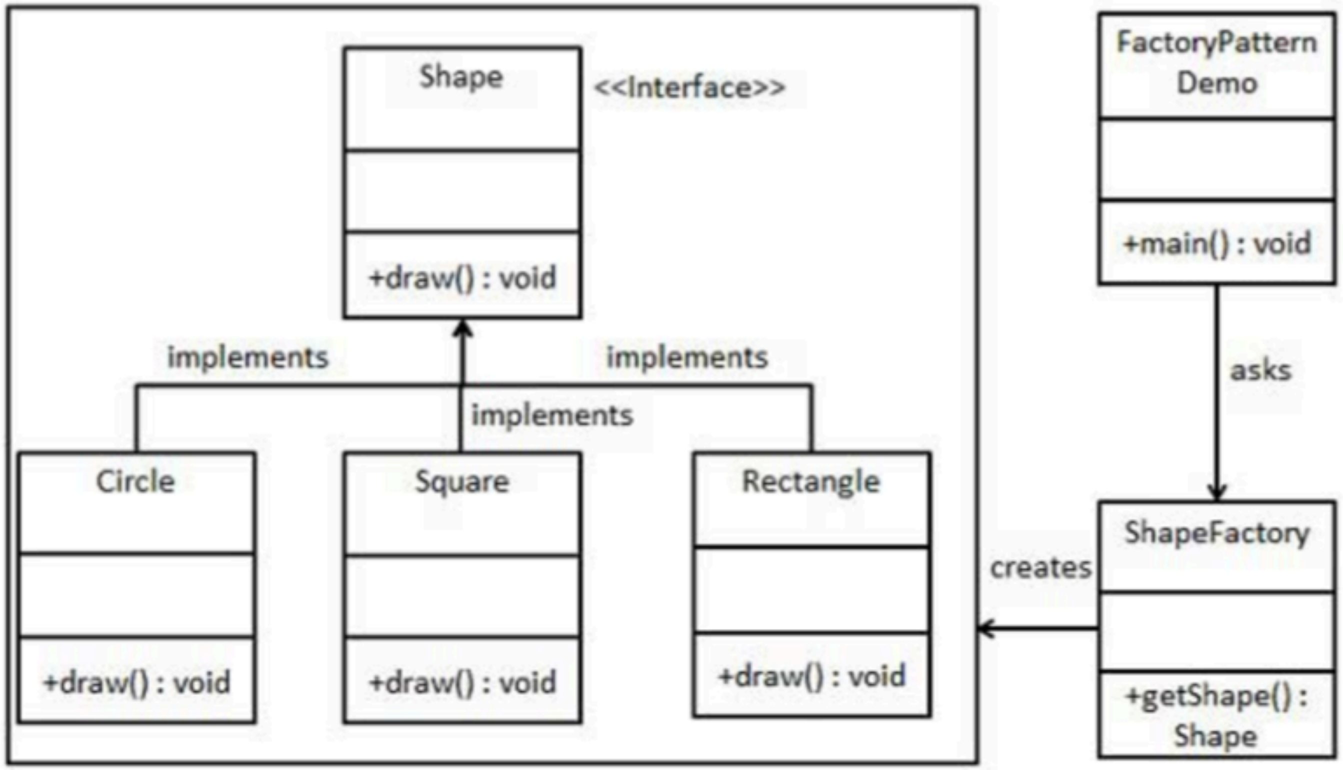
Typically, this is done by:

declaring all constructors of the class to be private;
and
providing a static method that returns a reference to the instance.

Factory Design Pattern

In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

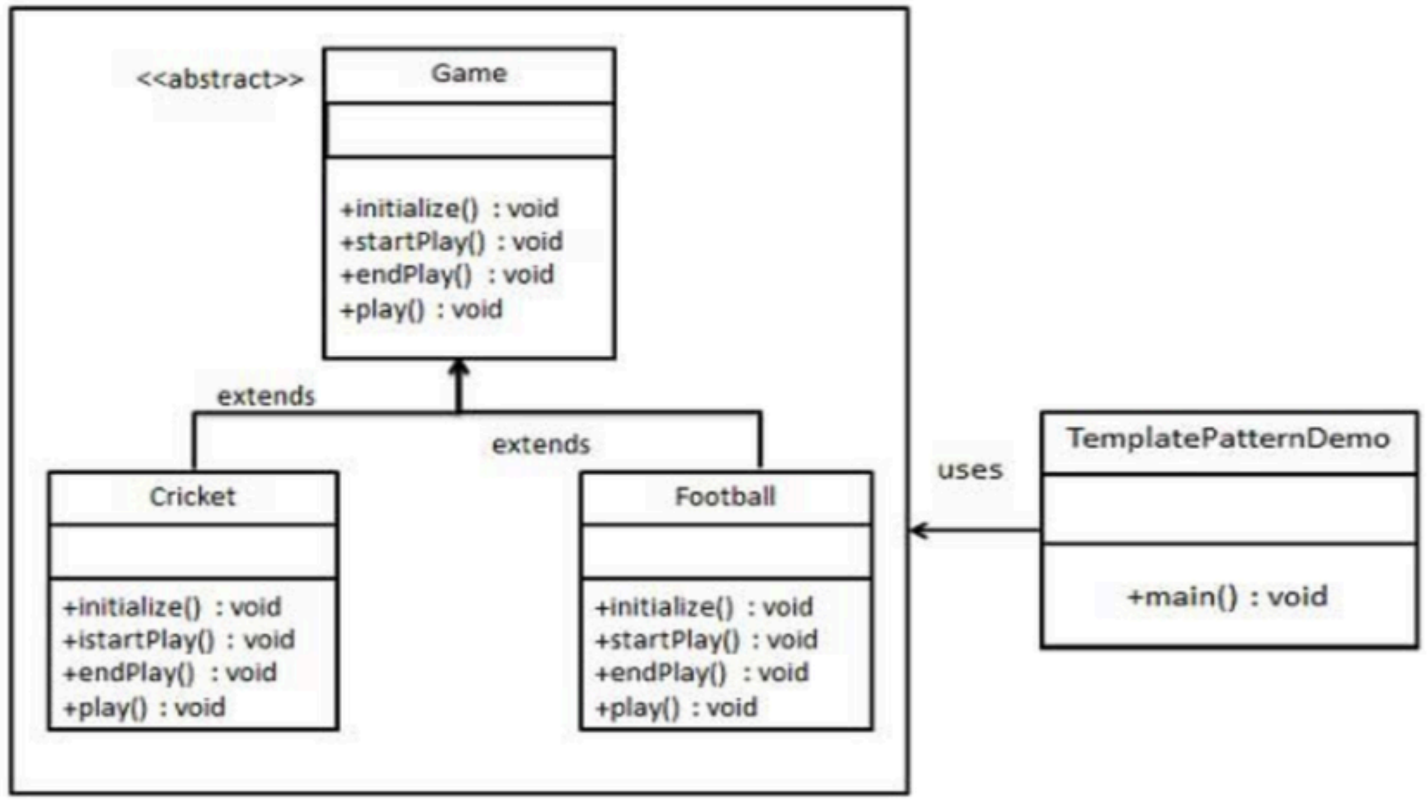
FactoryPatternDemo, our demo class will use *ShapeFactory* to get a *Shape* object. It will pass information (*CIRCLE* / *RECTANGLE* / *SQUARE*) tc *ShapeFactory* to get the type of object it needs.



```java
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;
   }
}
```

```java
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();
```

Template Pattern



In Template pattern, an abstract class exposes defined way(s)/template(s) to execute its methods. Its subclasses can override the method implementation as per need but the invocation is to be in the same way as defined by an abstract class.

```
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!");
   }
}
```

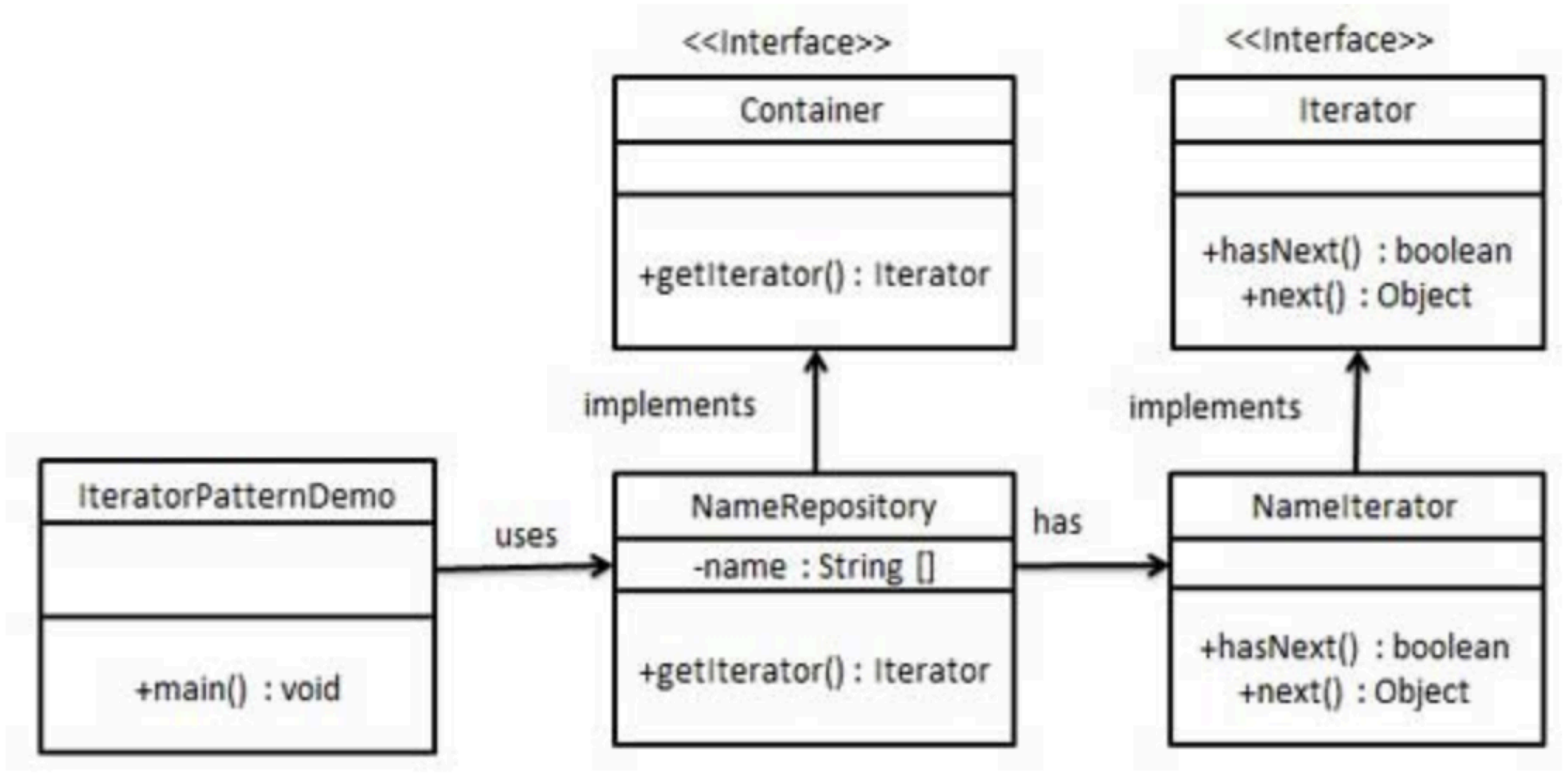
```
public abstract class Game {
   abstract void initialize();
   abstract void startPlay();
   abstract void endPlay();

   //template method
   public final void play(){

      //initialize the game
      initialize();

      //start game
      startPlay();

      //end game
      endPlay();
   }
}
```

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

      Game game = new Cricket();
      game.play();
      System.out.println();
      game = new Football();
      game.play();
   }
}
```

Iterator Design Pattern

Goal - access the elements of an aggregate object sequentially without exposing its underlying representation (behavioral pattern)

- How can you loop over all objects in any collection? You don’t want to change client code when the collection changes. Want the same methods.
- Solution: 1) Have each class implement an interface, and 2) Have an interface that works with all collections
- Consequences: Can change collection class details without changing code to traverse the collection


<<Interface>>
Container
+getIterator() : Iterator
<<Interface>>
Iterator
+hasNext() : boolean
+next() : Object
implements
implements
IteratorPatternDemo
+main() : void
uses
NameRepository
-name : String []
+getIterator() : Iterator
has
NameIterator
+hasNext() : boolean
+next() : Object

Iterator.java

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

Container.java

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

NameRepository.java

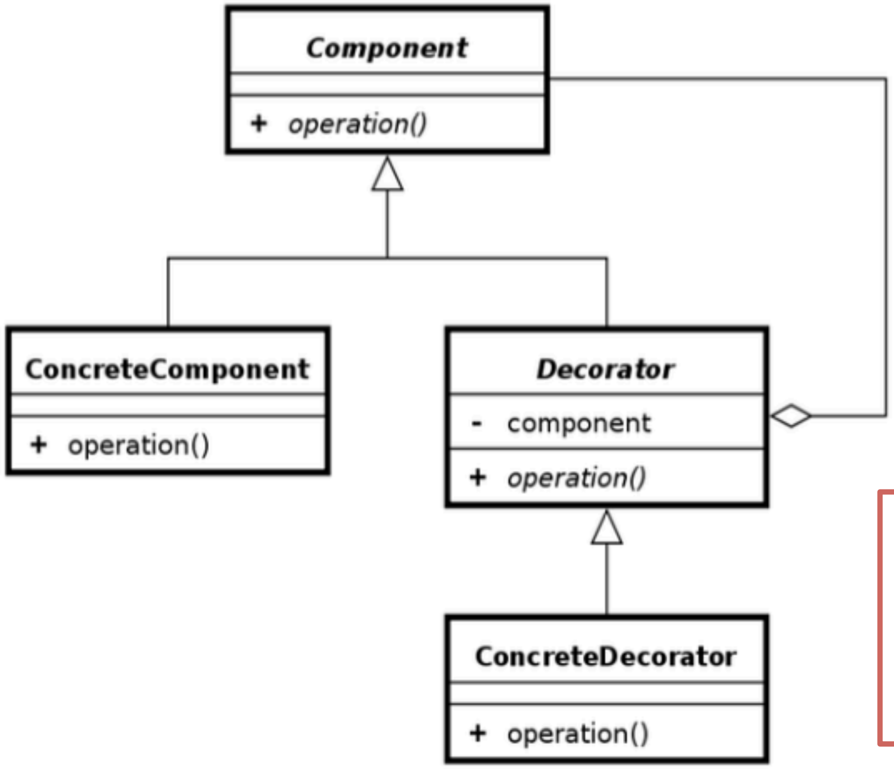
```
public class NameRepository implements Container {
   public String names[] = {"Robert" , "John" ,"Julie" , "Lora"};

   @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;
      }
   }
}
```

IteratorPatternDemo.java

```
public class IteratorPatternDemo {

   public static void main(String[] args) {
      NameRepository namesRepository = new NameRepository();

      for(Iterator iter = namesRepository.getIterator(); iter.hasNext();){
         String name = (String)iter.next();
         System.out.println("Name : " + name);
      }
   }
}
```

Decorator(Wrapper) Pattern

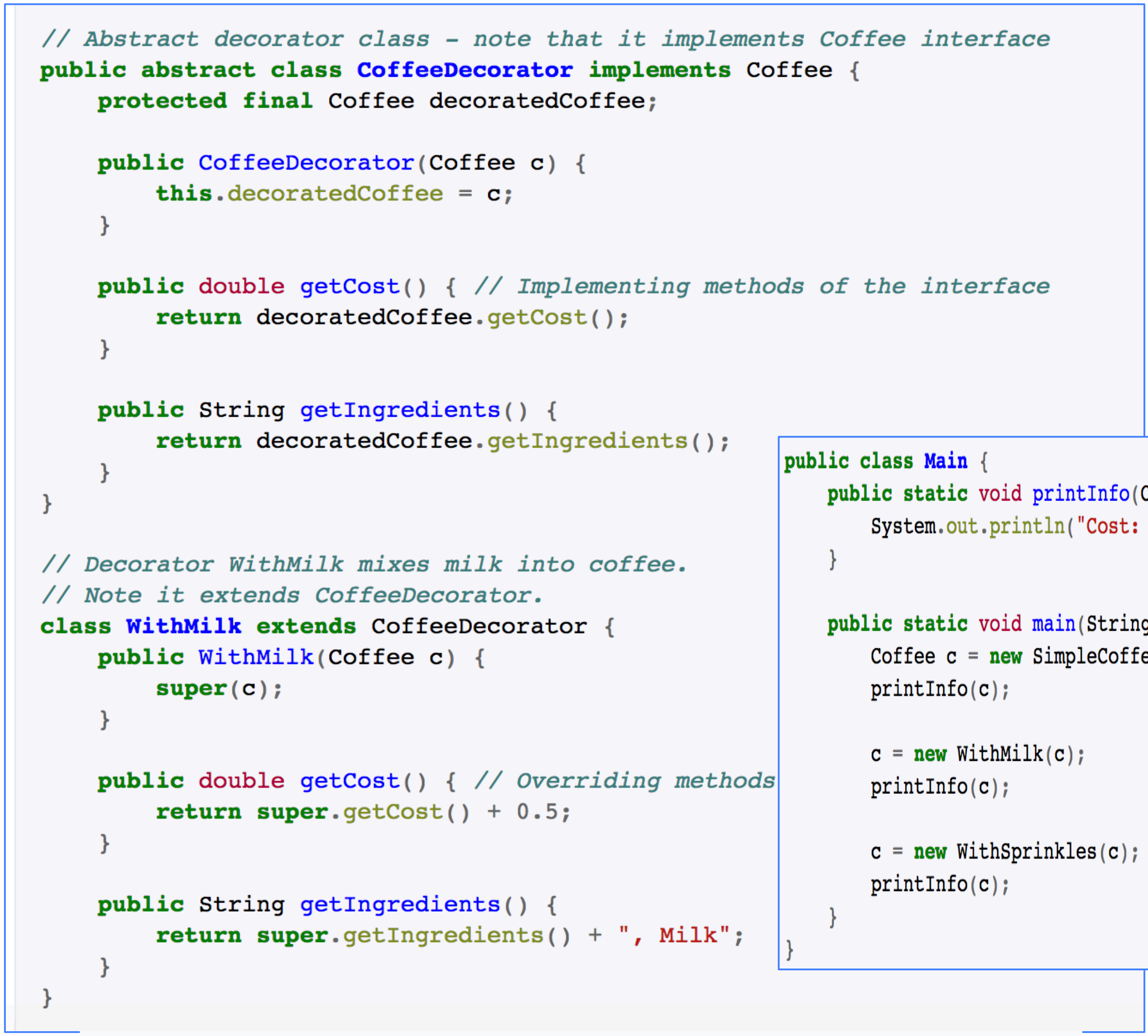
It is a design pattern that allows behavior to be added to an individual object, either statically or dynamically, without affecting the behavior of other objects from the same class



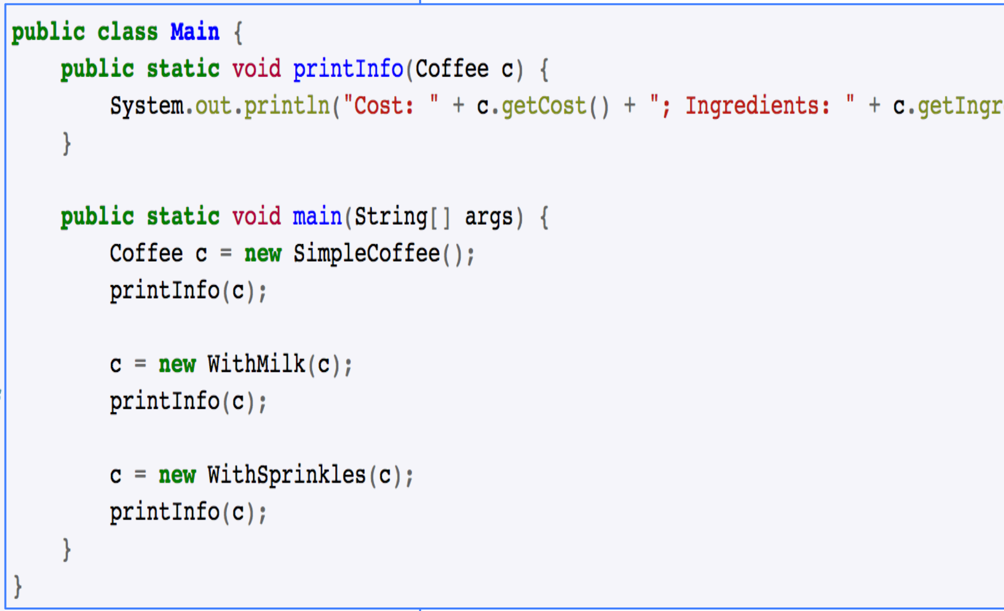
The decorator pattern can be used to extend (decorate) the functionality of a certain object statically, or in some cases at run-time, independently of other instances of the same class

Decorators provide a flexible alternative to sub classing to extend flexibility

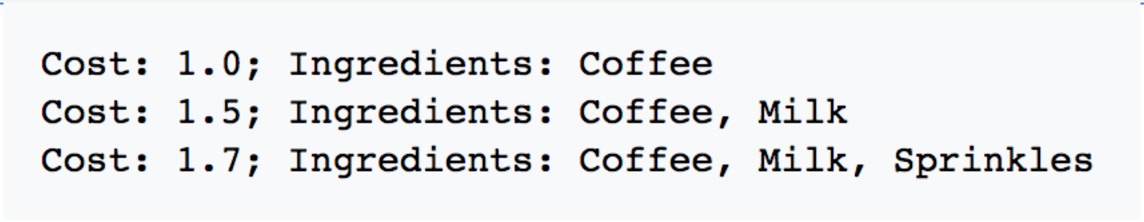
```java
public interface Coffee {
    public double getCost(); // Returns the cost
    public String getIngredients(); // Returns t
}

// Extension of a simple coffee without any extr
public class SimpleCoffee implements Coffee {
    @Override
    public double getCost() {
        return 1;
    }

    @Override
    public String getIngredients() {
        return "Coffee";
    }
}
```



```
// Abstract decorator class - note that it implements Coffee interface
public abstract class CoffeeDecorator implements Coffee {
    protected final Coffee decoratedCoffee;

    public CoffeeDecorator(Coffee c) {
        this.decoratedCoffee = c;
    }

    public double getCost() { // Implementing methods of the interface
        return decoratedCoffee.getCost();
    }

    public String getIngredients() {
        return decoratedCoffee.getIngredients();
    }
}

// Decorator WithMilk mixes milk into coffee.
// Note it extends CoffeeDecorator.
class WithMilk extends CoffeeDecorator {
    public WithMilk(Coffee c) {
        super(c);
    }

    public double getCost() { // Overriding methods
        return super.getCost() + 0.5;
    }

    public String getIngredients() {
        return super.getIngredients() + ", Milk";
    }
}
```



```
public class Main {
    public static void printInfo(Coffee c) {
        System.out.println("Cost: " + c.getCost() + "; Ingredients: " + c.getIngr
    }

    public static void main(String[] args) {
        Coffee c = new SimpleCoffee();
        printInfo(c);

        c = new WithMilk(c);
        printInfo(c);

        c = new WithSprinkles(c);
        printInfo(c);
    }
}
```



```
Cost: 1.0; Ingredients: Coffee
Cost: 1.5; Ingredients: Coffee, Milk
Cost: 1.7; Ingredients: Coffee, Milk, Sprinkles
```

Strategy Pattern

- A means to define a family of algorithms, encapsulate each one as an object, and make them interchangeable
- Create an abstract strategy class (or interface) and extend (or implement) it in numerous ways. Each subclass defines the same method names in different ways

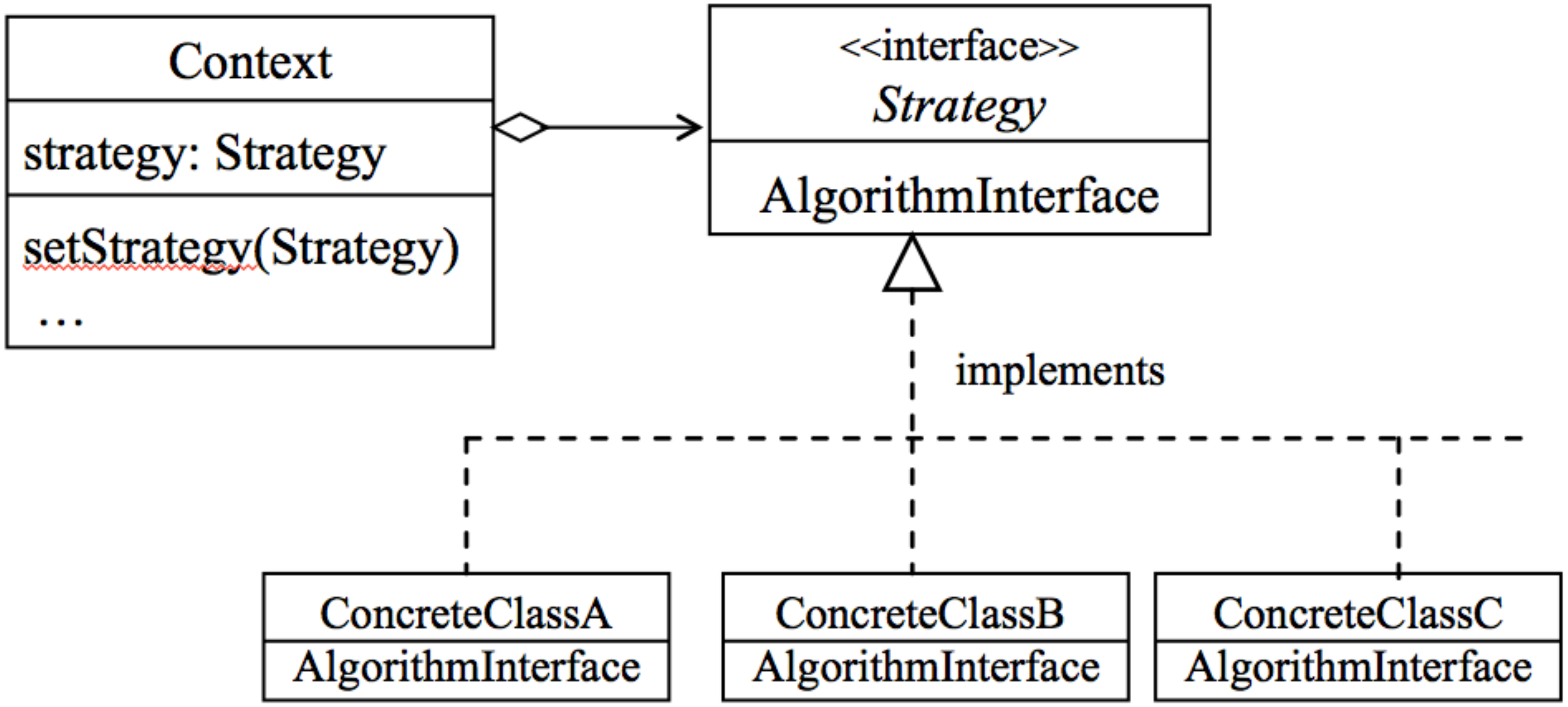
Strategy pattern is used when we have multiple algorithm for a specific task and client decides the actual implementation to be used at runtime.

Examples:
Layout managers
Different PacMan chase strategies

Example

```
this.setLayout(new FlowLayout());
this.setLayout(new GridLayout());
```

- In Java, a container HAS-A layout manager
  - There is a default
  - You can change a container's layout manager with a **`setLayout`** message


Context
strategy: Strategy
setStrategy(Strategy)
…
<<interface>>
Strategy
AlgorithmInterface
implements
ConcreteClassA
AlgorithmInterface
ConcreteClassB
AlgorithmInterface
ConcreteClassC
AlgorithmInterface

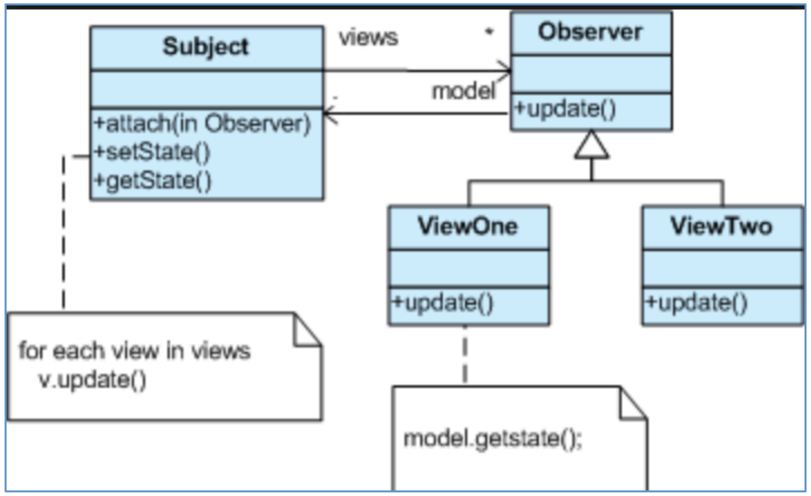
Observer Pattern

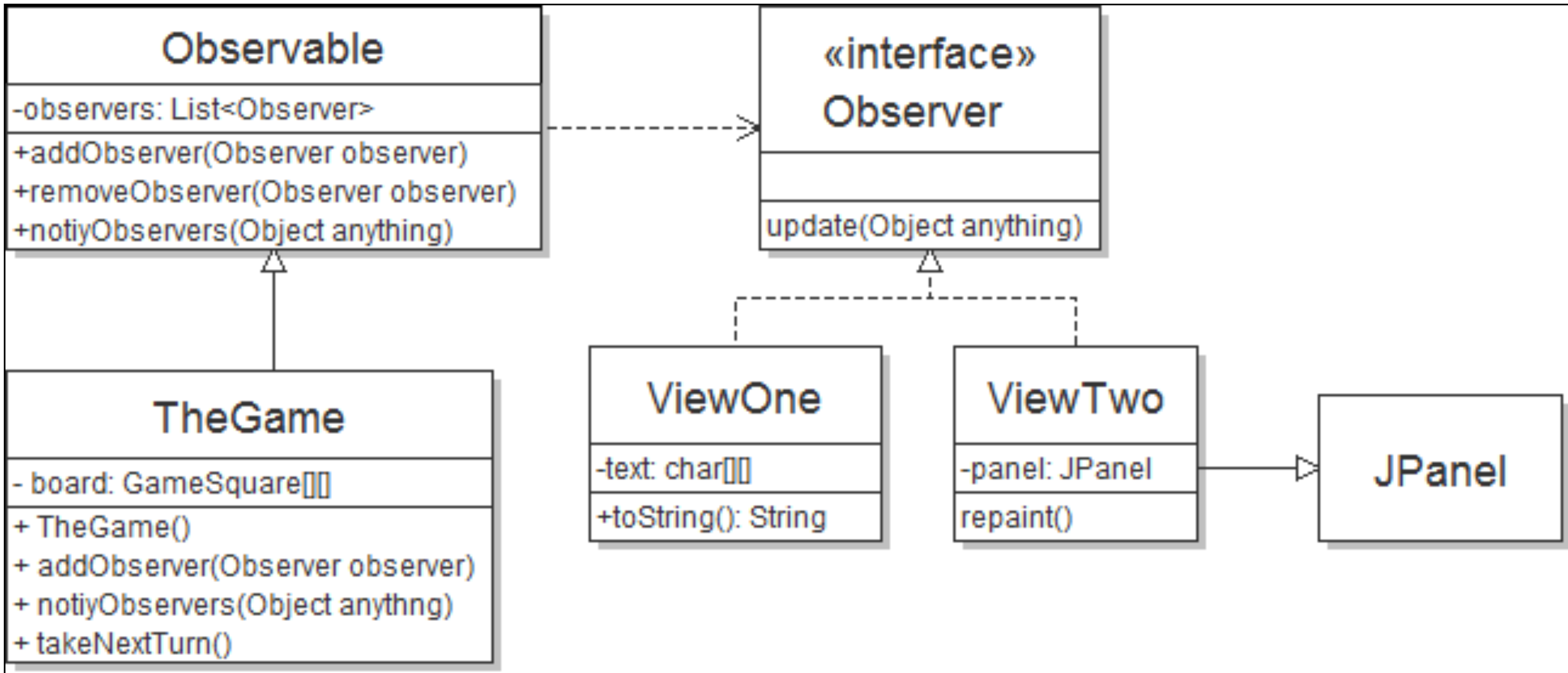
- One object stores a list of observers that are updated when the state of the object is changed.
- File Explorer (or Finders) are registered observers (the view) of the file system (the model).
- You open several finders to view file system and delete a file

Problem: Need to notify a changing number of objects that something has changed
Solution: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically

Examples:
Send a newspaper to all who subscribe
People add and drop subscriptions, when a new version comes out, it goes to all currently described

Observer design pattern is useful when you are interested in the state of an object and want to get notified whenever there is any change.


Subject
+attach(in Observer)
+setState()
+getState()
views
*
model
Observer
+update()
ViewOne
+update()
ViewTwo
+update()
for each view in views
v.update()
model.getstate();


Observable
-observers: List<Observer>
+addObserver(Observer observer)
+removeObserver(Observer observer)
+notiyObservers(Object anything)
«interface»
Observer
update(Object anything)
TheGame
- board: GameSquare[][]
+ TheGame()
+ addObserver(Observer observer)
+ notiyObservers(Object anythng)
+ takeNextTurn()
ViewOne
-text: char[][]
+toString(): String
ViewTwo
-panel: JPanel
repaint()
JPanel

Creational Patterns

- **Abstract Factory**:
  - Factory for building related objects
- **Builder**:
  - Factory for building complex objects incrementally
- **Factory Method**:
  - Method in a derived class creates associates
- **Prototype**:
  - Factory for cloning new instances from a prototype
- **Singleton**:
  - Factory for a singular (sole) instance

Structural Pattern

- **Adapter:**
  - Translator adapts a server interface for a client
- **Bridge**:
  - Abstraction for binding one of many implementations
- **Composite**:
  - Structure for building recursive aggregations
- **Decorator**:
  - Decorator extends an object transparently
- **Facade**:
  - Simplifies the interface for a subsystem
- **Flyweight**:
  - Many fine-grained objects shared efficiently.
- **Proxy**:
  - One object approximates another

Behavioral Patterns

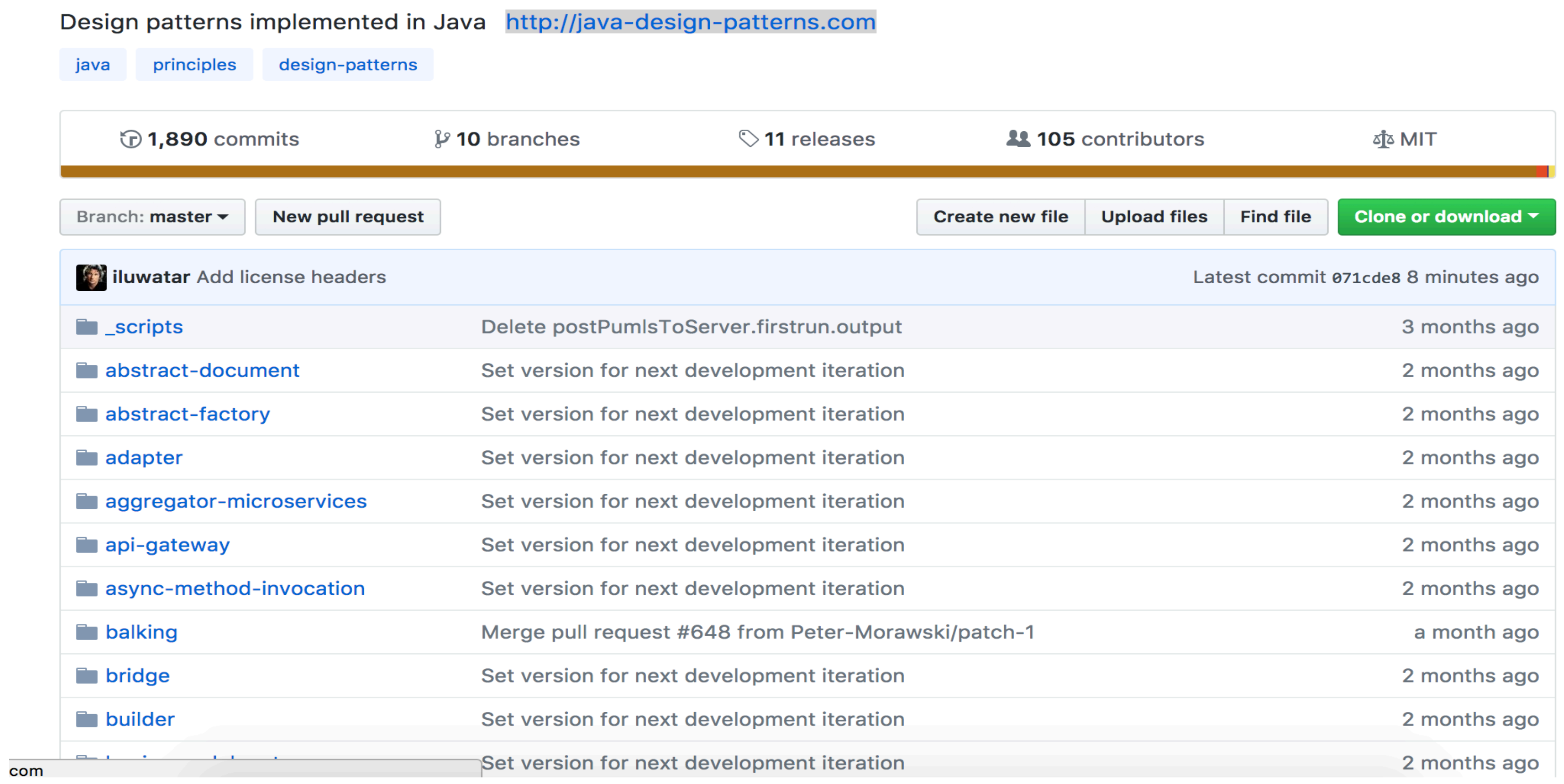
- **Chain of Responsibility**:
  - Request delegated to the responsible service provider
- **Command**:
  - Request or Action is first-class object, hence re-storable
- **Iterator**:
  - Aggregate and access elements sequentially
- **Interpreter**:
  - Language interpreter for a small grammar
- **Mediator**:
  - Coordinates interactions between its associates
- **Memento**:
  - Snapshot captures and restores object states privately

Behavioral Patterns

- **Observer**:
  - Dependents update automatically when subject changes
- **State**:
  - Object whose behavior depends on its state
- **Strategy**:
  - Abstraction for selecting one of many algorithms
- **Template Method**:
  - Algorithm with some steps supplied by a derived class
- **Visitor**:
  - Operations applied to elements of a heterogeneous object structure

A lot more examples…

- https://github.com/iluwatar/java-design-patterns



Design patterns implemented in Java http://java-design-patterns.com

java principles design-patterns

1,890 commits | 10 branches | 11 releases | 105 contributors | MIT

Branch: master | New pull request | Create new file | Upload files | Find file | Clone or download

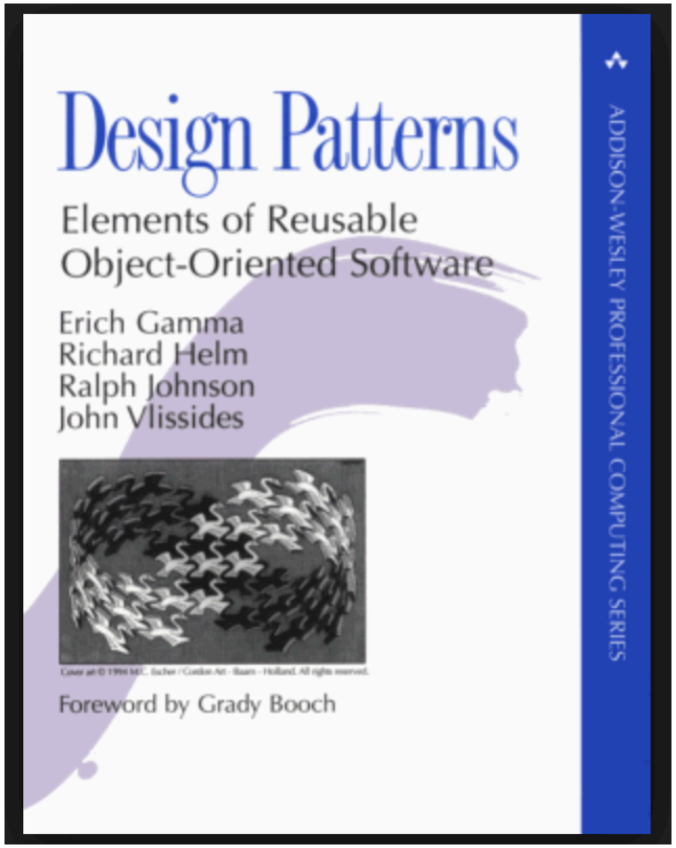
iluwatar Add license headers — Latest commit 071cde8 8 minutes ago

_scripts	Delete postPumlsToServer.firstrun.output	3 months ago
abstract-document	Set version for next development iteration	2 months ago
abstract-factory	Set version for next development iteration	2 months ago
adapter	Set version for next development iteration	2 months ago
aggregator-microservices	Set version for next development iteration	2 months ago
api-gateway	Set version for next development iteration	2 months ago
async-method-invocation	Set version for next development iteration	2 months ago
balking	Merge pull request #648 from Peter-Morawski/patch-1	a month ago
bridge	Set version for next development iteration	2 months ago
builder	Set version for next development iteration	2 months ago
	Set version for next development iteration	2 months ago

com

Best book ever…

Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Publishing Company, 1994


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Why Study Patterns?

- Reuse tried, proven solutions
  - No need to reinvent the wheel
- Establish common terminology
  - Design patterns provide a common point of reference
  - Easier to say, “We could use Strategy here.”
- Provide a higher level prospective
  - Frees us from dealing with the details too early
- Using design patterns makes software systems easier to change—more maintainable
- Helps increase the understanding of basic object-oriented design principles

Conclusion

- Design patterns enable large-scale reuse of software architectures and also help document systems
- Patterns explicitly capture expert knowledge and design tradeoffs and make it more widely available
- Patterns help improve developer communication