Software design pattern

Tarun Mohandas Daryanani
Abián Torres Torres

Content

- **Introduction**
  - Objetives
  - Categories
  - Pattern Templates
- **Creational Patterns**
  - Examples
- **Structure Patterns**
  - Examples
- **Behavoir Patterns**

Introduction

- What are software design patterns?
- Objectives of design patterns
- Categories
  - Architecture patterns
  - Design patterns
  - Dialects
- Pattern templates

Pattern goals

- Pattern design allow:
- Provide catalogue of reusables elements in the software design pattern.
- Avoid repetition for solution search to problems already known and previously solved.
- Formalize a common vocabulary between designers.
- Standardize the way the design is done.
- Facilitate the learning of new generation of designers by condensing existing knowledge.

Pattern categories

- **Architecture patterns**: Provide a solution to organizational scheme in software
- **Design patterns**: Provide a scheme to define the structures in our design or their relations to build software.
- **Dialects**: Low level patterns specific for a programming language of specific environment.

“Efforts have also been made to codify design patterns in particular domains, including use of existing design patterns as well as domain specific design patterns. Examples include user interface design patterns, information visualization, secure design, "secure usability", Web design and business model design. The annual Pattern Languages of Programming Conference proceedings include many examples of domain-specific patterns.”

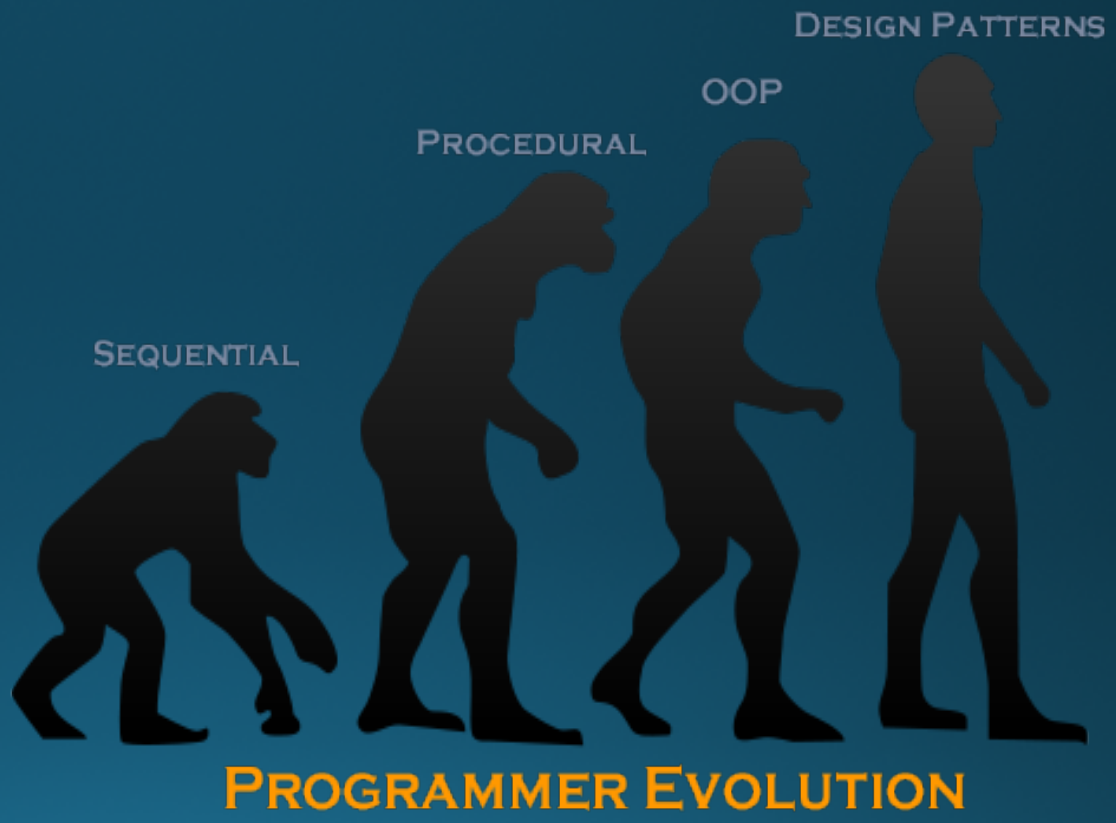
Pattern templates

- ***Pattern name*:** Standard name for the pattern recognized by the comunity
- ***Patter category***: creational, structural or behaviour.
- ***Intention***: What problema does it solve?
- ***Motivation***: Scenario of the example for the pattern application.
- ***Aplicability***: Common uses and criteria to apply the pattern.
- **Participants:** Specify and enumerate the abstract entitiesthat participate.
- ***Structure***: Diagrams and clases to explain the pattern.
- ***Colaborations***: Explanation of the interrelations between the participants.
- ***Consecuencies***: Positive and negatives consecuencies of the pattern application.
- ***Implementation***: Appropiate comentaries and tecniques for the pattern implementation.
- ***Example codes***: Source code for the pattern implementation.
- ***Known uses*.**

http://www.oodesign.com/
https://www.tutorialspoint.com/design_pattern/

Design patterns

- Design patterns can speed up the development process by providing tested, proven development paradigms. Effective software design requires considering issues that may not become visible until later in the implementation.



Design patterns: Types

- **Creational patterns**
  - Patterns that solve problems with instances creation.
- **Structural patterns**
  - Patterns that provide a solution to clases or objects composition.
- **Behaviour patterns**
  - Provide a solution for responsability and interation of classes and objects, as well as algorithms.

Creational patterns (GOF)

- Singleton
- Factory Method
- Abstract factory
- Builder
- Prototype
- Object Pool

"In software engineering, creational design patterns are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation could result in design problems or in added complexity to the design. Creational design patterns solve this problem by somehow controlling this object creation."

Creational patterns: Singleton

- **Motivation**:

  Sometimes it's important to have only one instance for a class. For example, in a system there should be only one window manager

  (or only a file system or only a print spooler). Usually singletons are used for centralized management of internal or external resources and they provide a global point of access to themselves.

Creational patterns: Singleton

- **Applicability**:

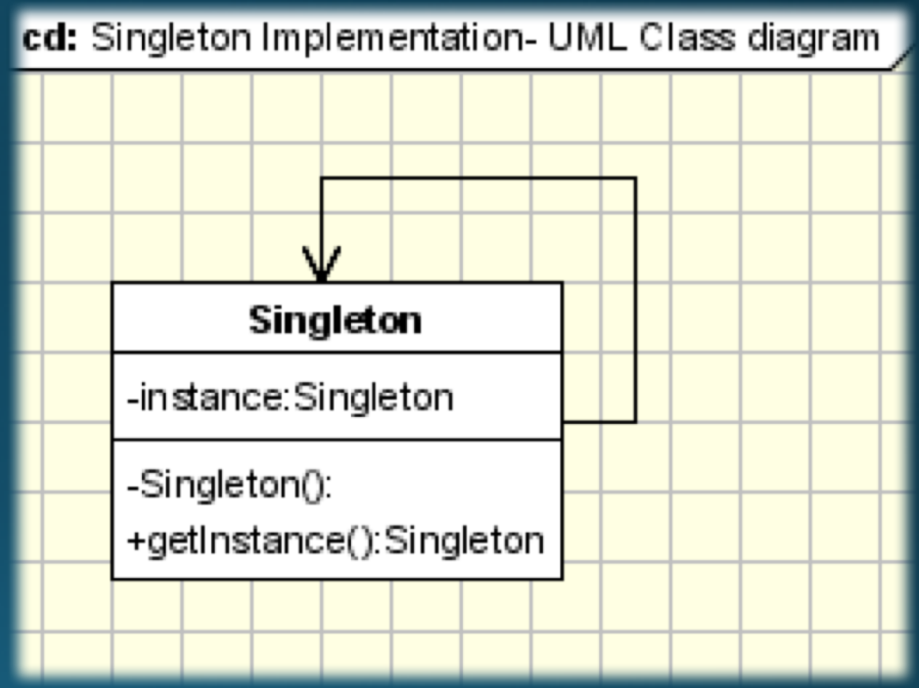
  According to the definition the singleton pattern should be used when there must be exactly one instance of a class, and when it must be accessible to clients from a global access point.

- **Examples**:
  - Accessing resources in shared mode
  - Configuration Classes

Creational patterns: Singleton

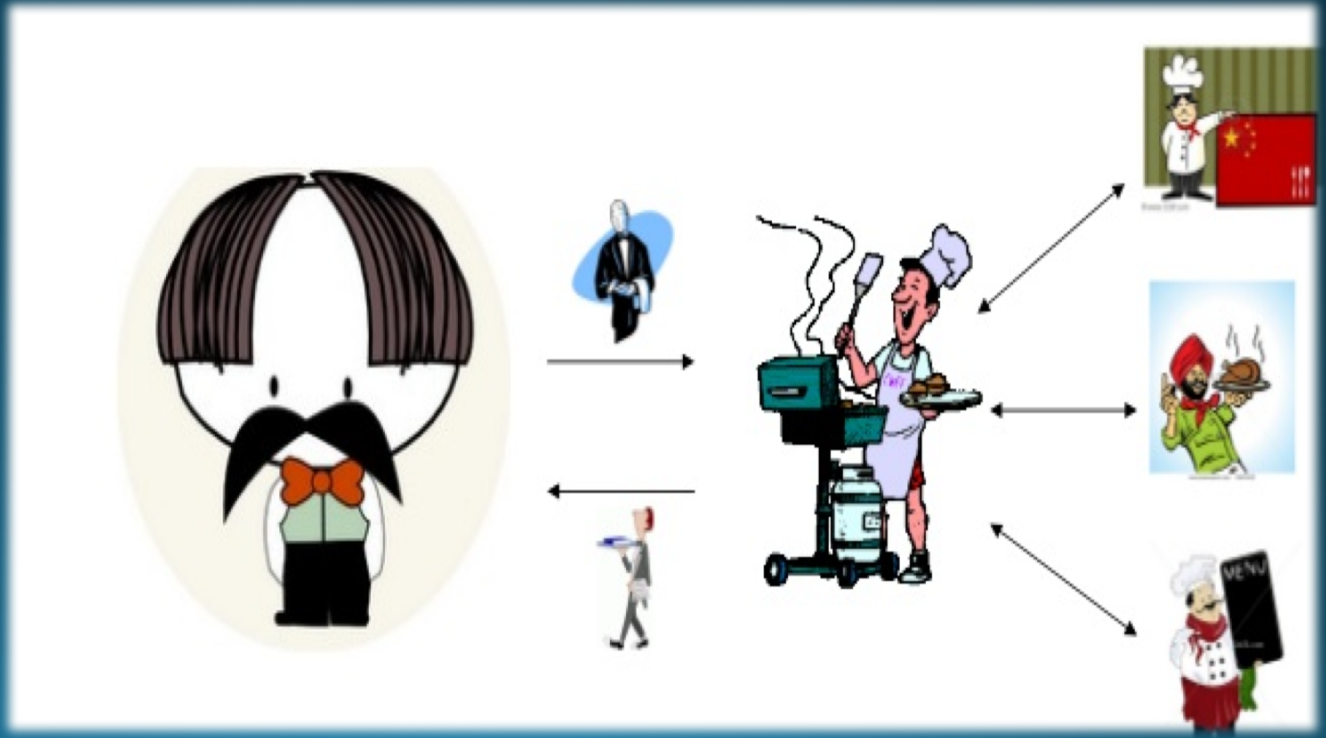
- **Implementation**

  The implementation involves a static member in the "Singleton" class, a private constructor and a static public method that returns a reference to the static member.



Creational patterns: Abstract Factory

- **Motivation**
  - Programmers all over the world are trying to avoid the idea of adding code to existing classes in order to make them support encapsulating more general information.
  - Example for Window.



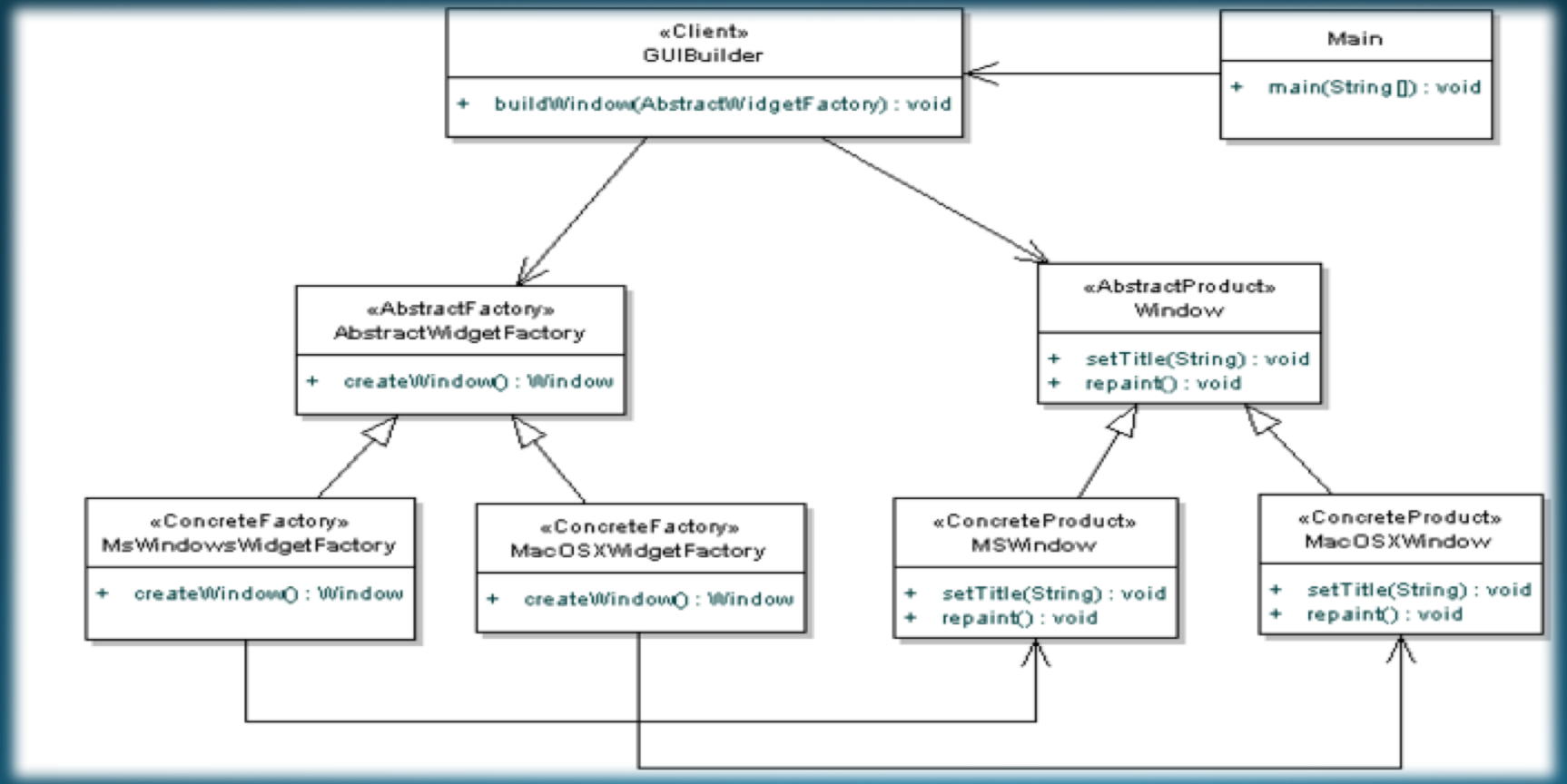
Creational patterns: Abstract Factory

- **Applicability**
  - the system is or should be configured to work with multiple families of products.
  - a family of products is designed to work only all together.

- **Examples**
  - Phone Number Example
  - Pizza Factory Example

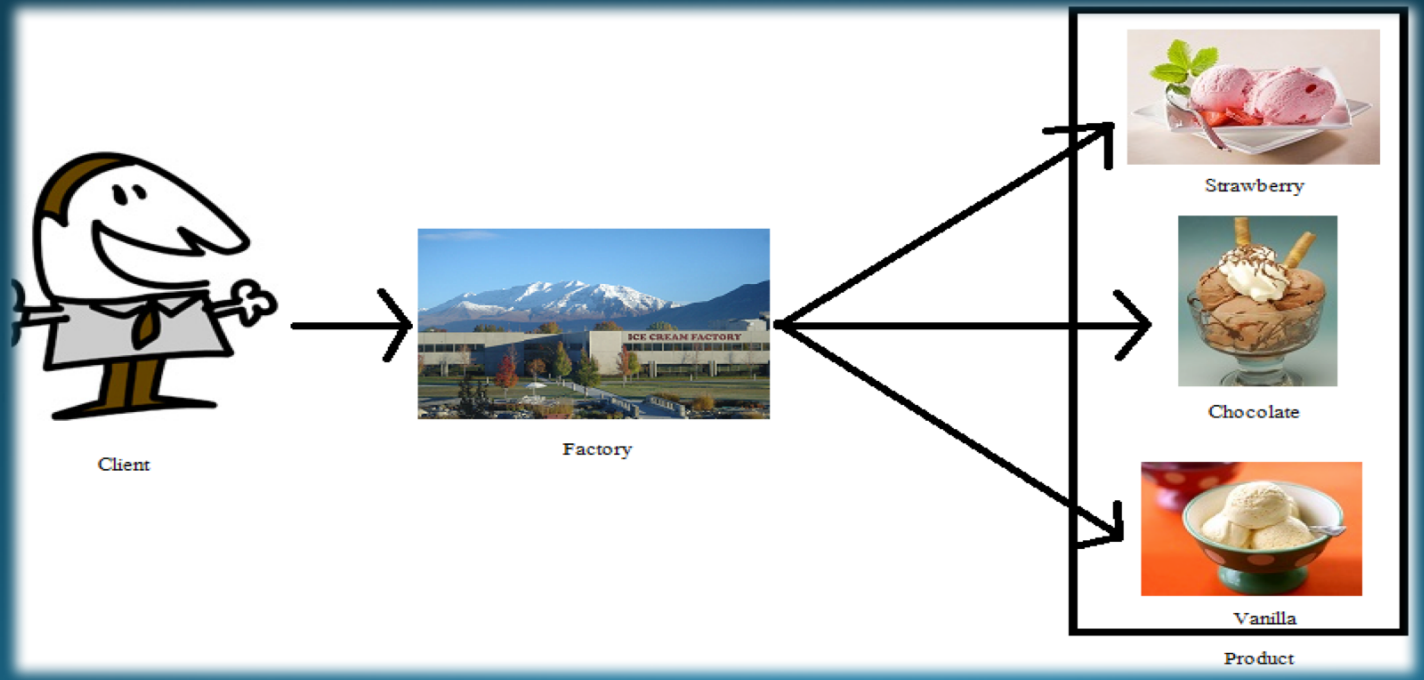
Creational patterns: Abstract Factory

- Implementation



Creational patterns: Factory Method

- **Motivation**
  - It defines an interface for creating an object, but leaves the choice of its type to the subclasses, creation being deferred at run-time. Also known as Virtual Constructor. It works in the same way as libraries.



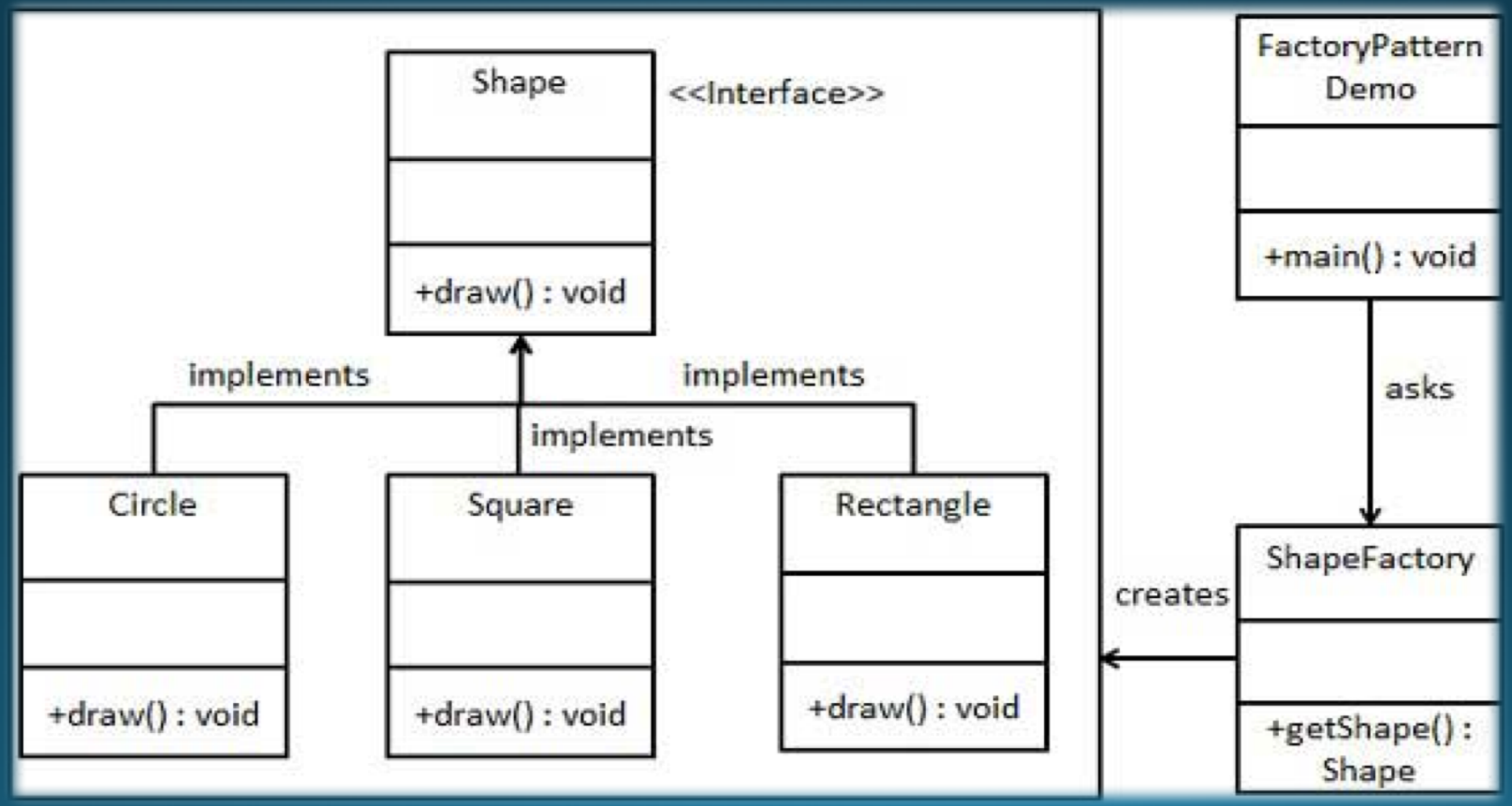
Creational patterns: Factory Method

- Applicability
  - When a class doesn't know what sub-classes will be required to create
  - When a class wants that its sub-classes specify the objects to be created.
  - When the parent classes choose the creation of objects to its sub-classes.

- Factories, specifically factory methods, are common in toolkits and frameworks, where library code needs to create objects of types that may be subclassed by applications using the framework.

Creational patterns: Factory Method

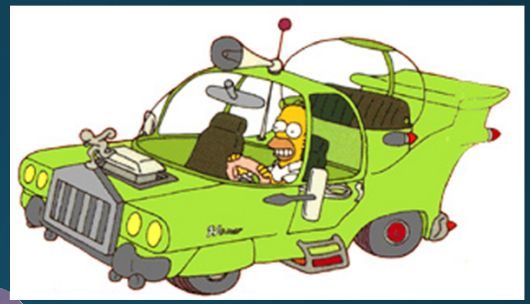
- Implementation:

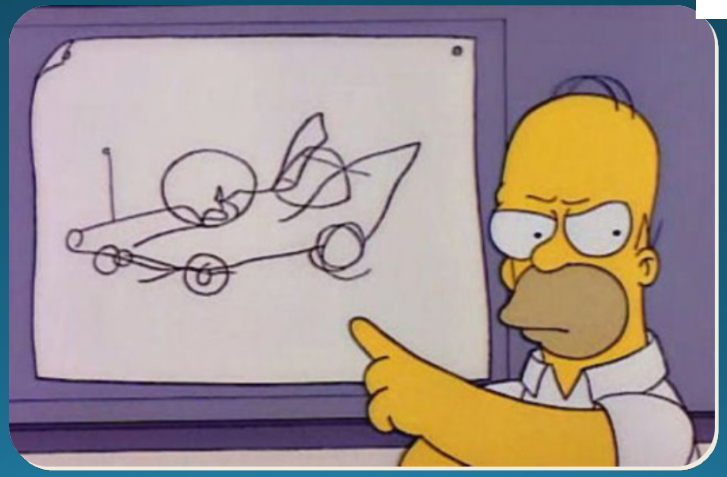


Creational patterns: Builder

- **Differences between Factory Method and Builder:**
  - Builder pattern makes the object step by step and finally returns the result object.
  - Factory Method directly makes all the operations and automatically returns the final object.

Abstraction

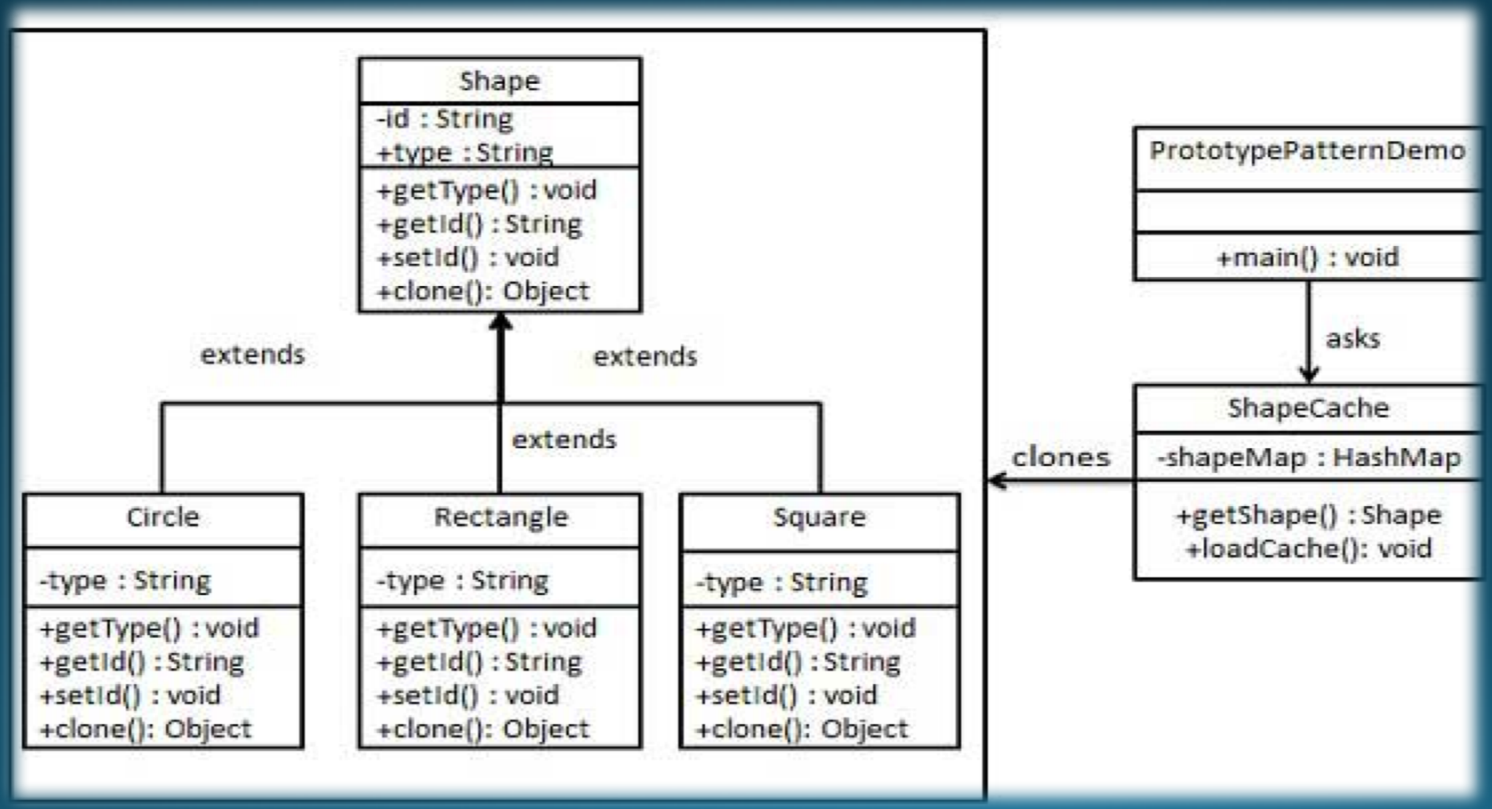




Emphasis on "what does it do?" more than "how it does?"

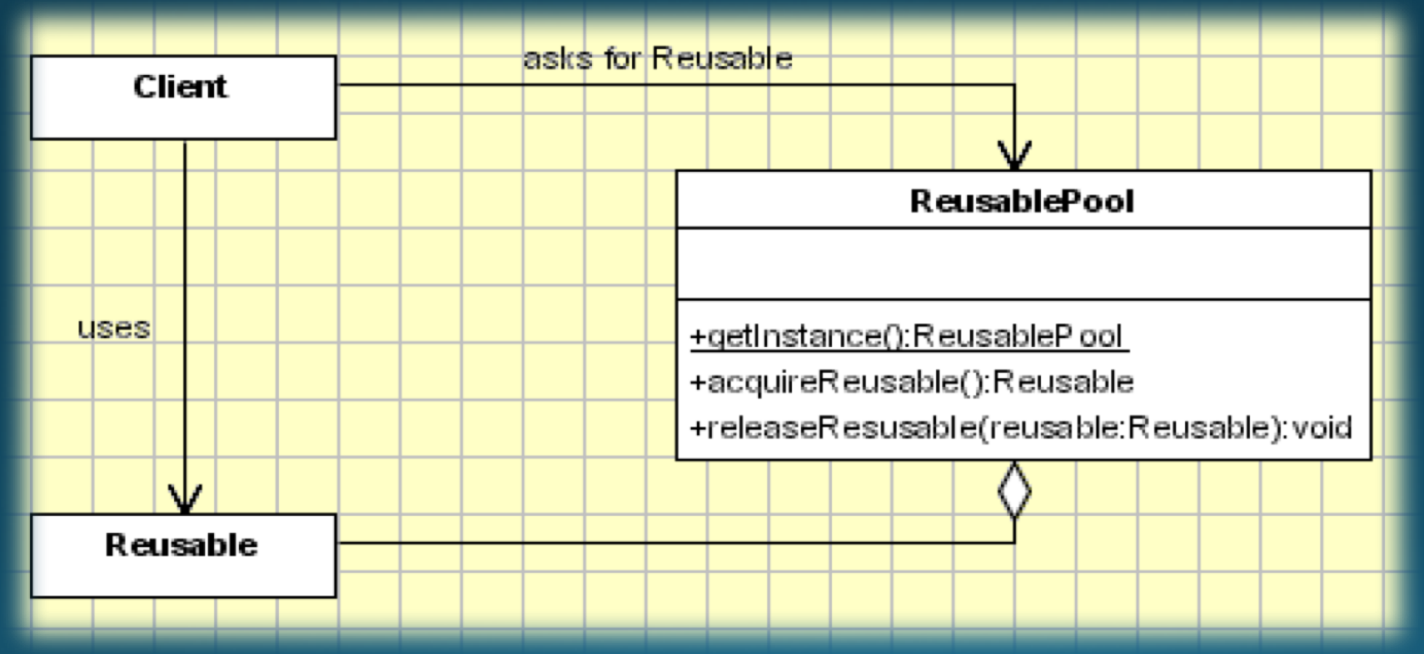
Creational patterns: Prototype

- **Short Description:**



Creational patterns: Object Pool

- Short Description:



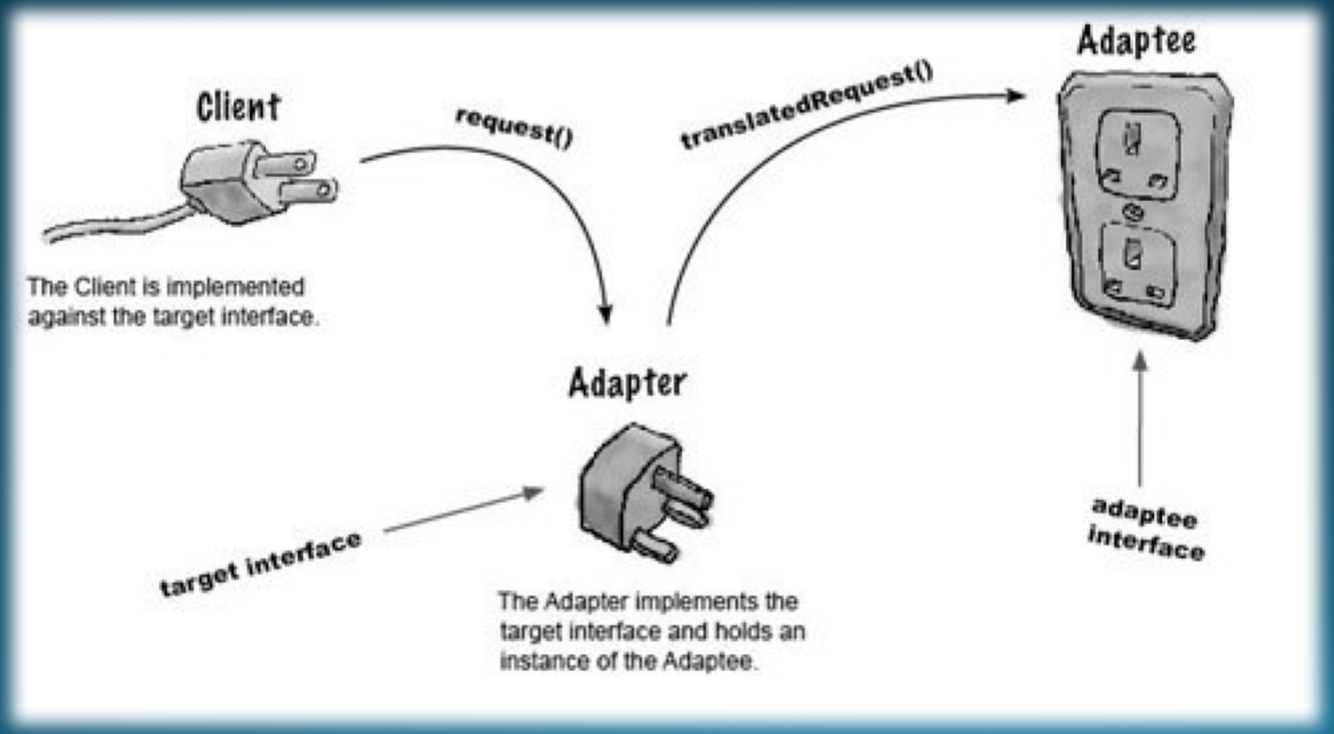
Structural patterns (GOF)

- **Adapter**
- **Bridge**
- Composite
- **Decorator**
- Flyweight
- Memento
- Proxy

“In software engineering, structural design patterns are design patterns that ease the design by identifying a simple way to realize relationships between entities.”

Structural patterns: Adapter

- Motivation:
  - Adapter pattern works as a bridge between two incompatible interfaces. This type of design pattern comes under structural pattern as this pattern combines the capability of two independent interfaces.

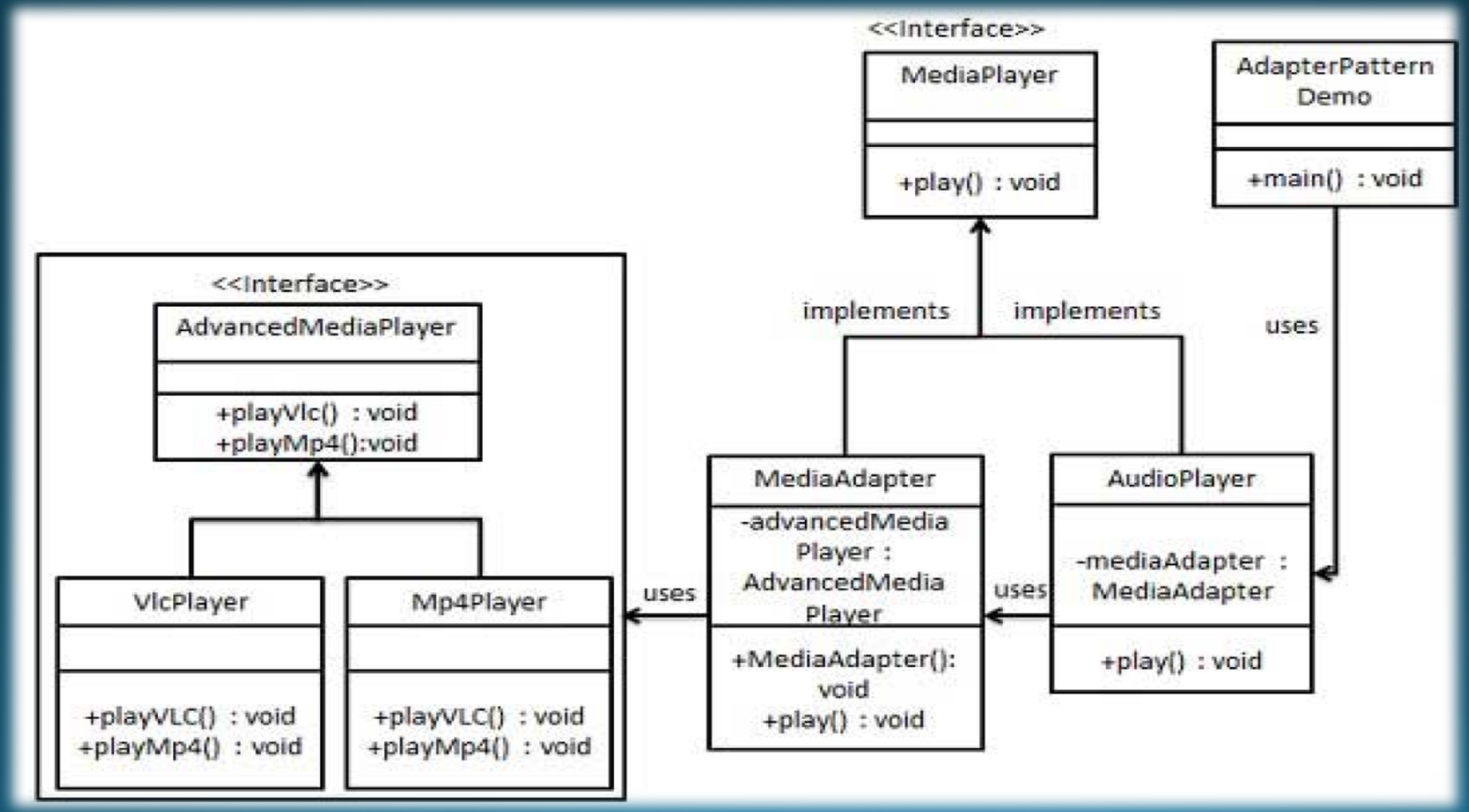


Structural patterns: Adapter

- **Applicability**:
  - Use the Adapter pattern when you want to use an existing class, and its interface does not match the one you need.
  - You want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don’t necessarily have compatible interfaces.

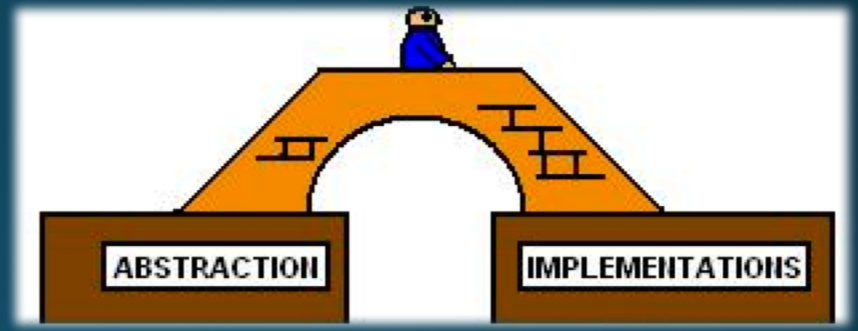
Structural patterns: Adapter

- Implementation:



Structural patterns: Bridge

- **Motivation**:
  - This pattern involves an interface which acts as a bridge which makes the functionality of concrete classes independent from interface implementer classes. Both types of classes can be altered structurally without affecting each other.



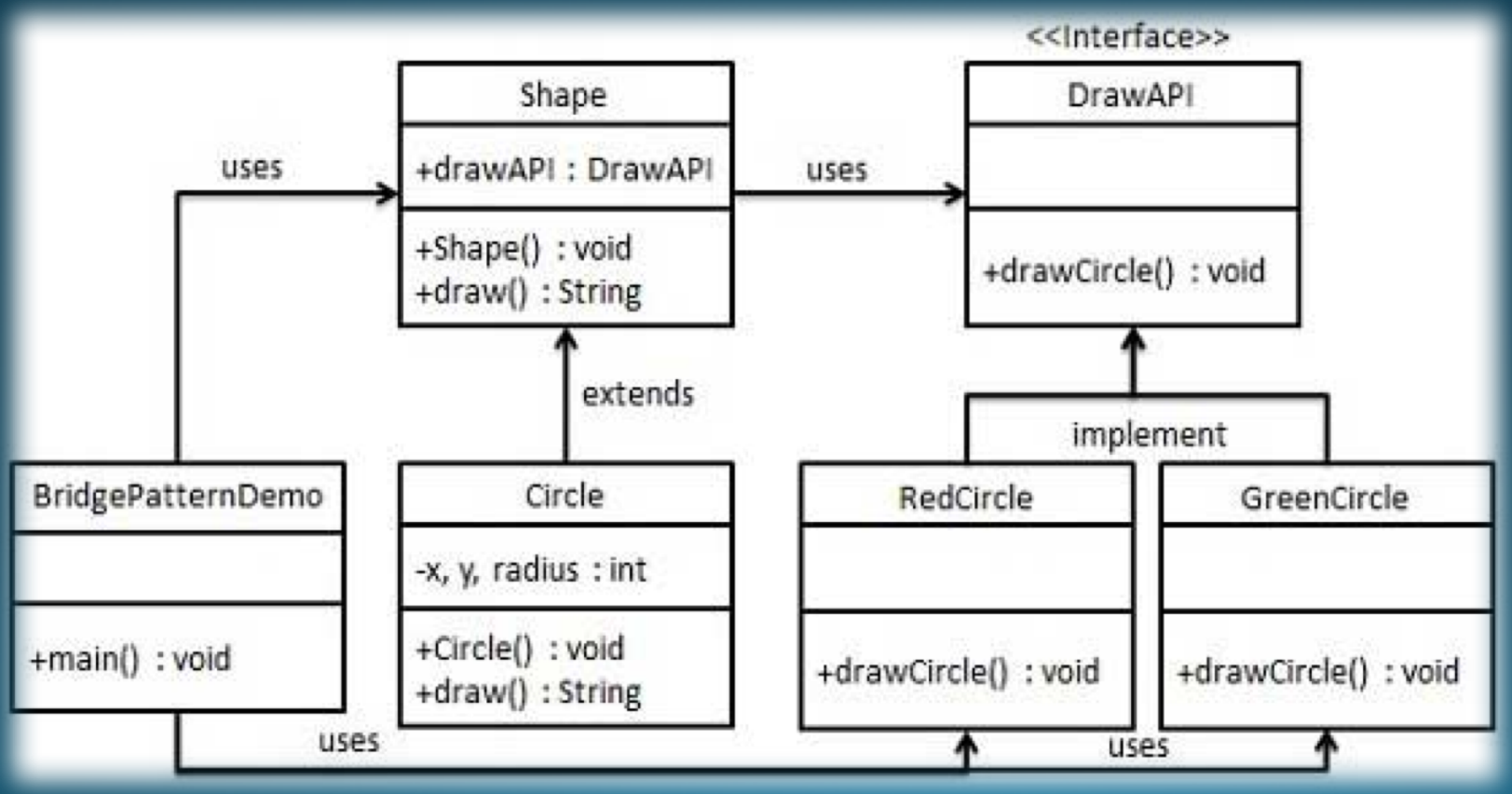
Structural patterns: Bridge

- **Applicability**:
  - Used when we have independent classes but need to cooperate. So these classes can change, but they will always be able to communicate and depend on each other.

"Adapter makes things work after they're designed.
Bridge makes them work before they are. [GoF, p219]"

Structural patterns: Bridge

- Implementation



Structural patterns: Decorator

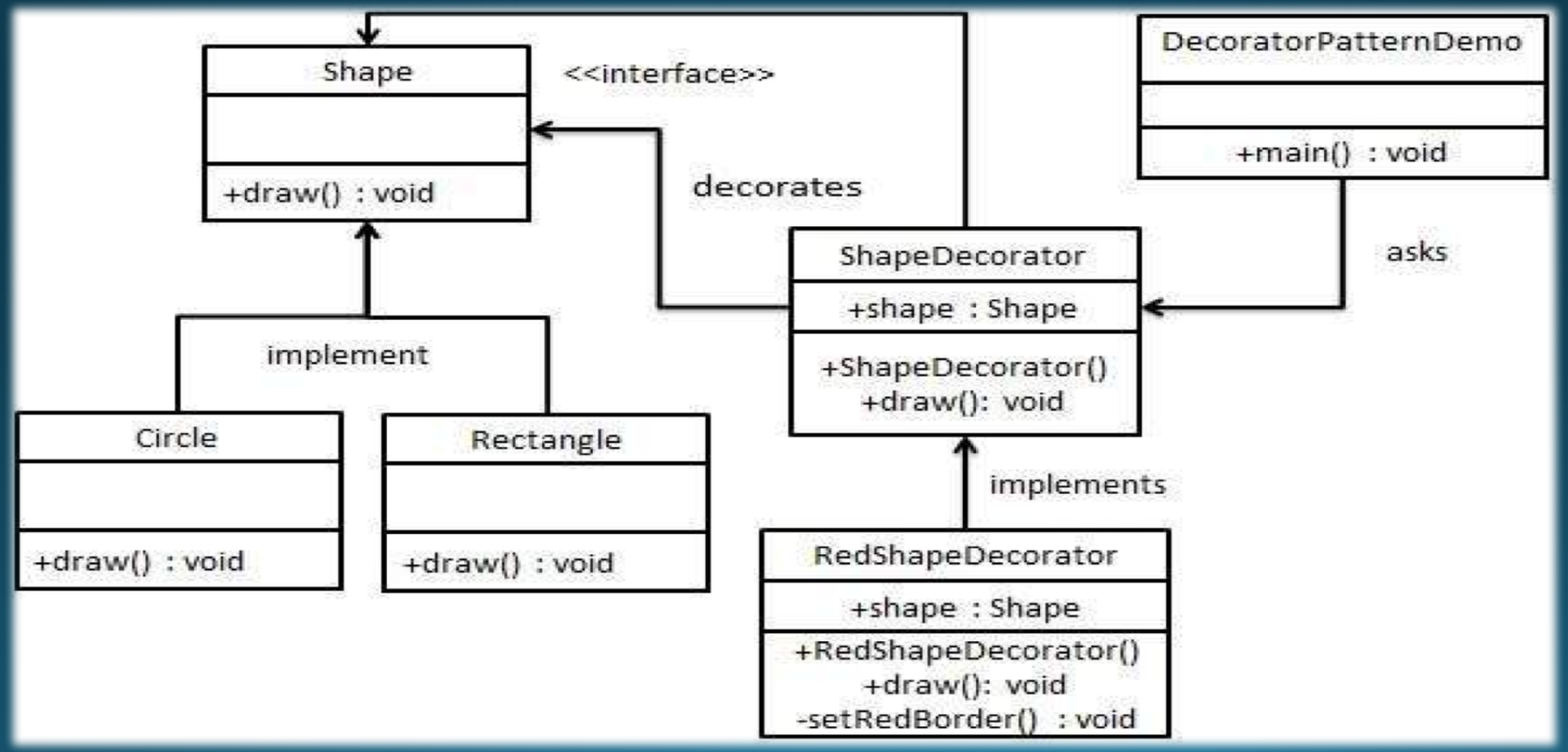
- Motivation
  - Decorator pattern allows a user to add new functionality to an existing object without altering its structure. This type of design pattern comes under structural pattern as this pattern acts as a wrapper to existing class.

Structural patterns: Decorator

- **Applicability**:
  - Used when we have independent classes but need to cooperate. So these classes can change, but they will always be able to communicate and depend on each other.

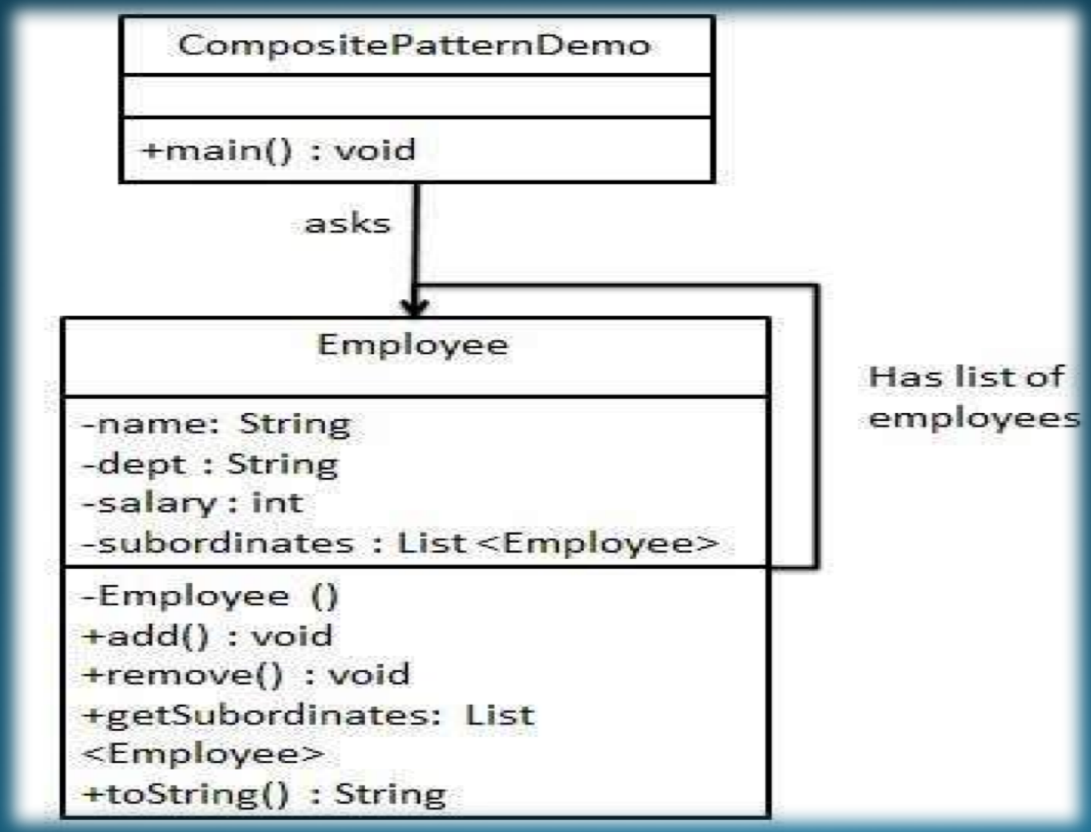
Structural patterns: Decorator

- Implementation



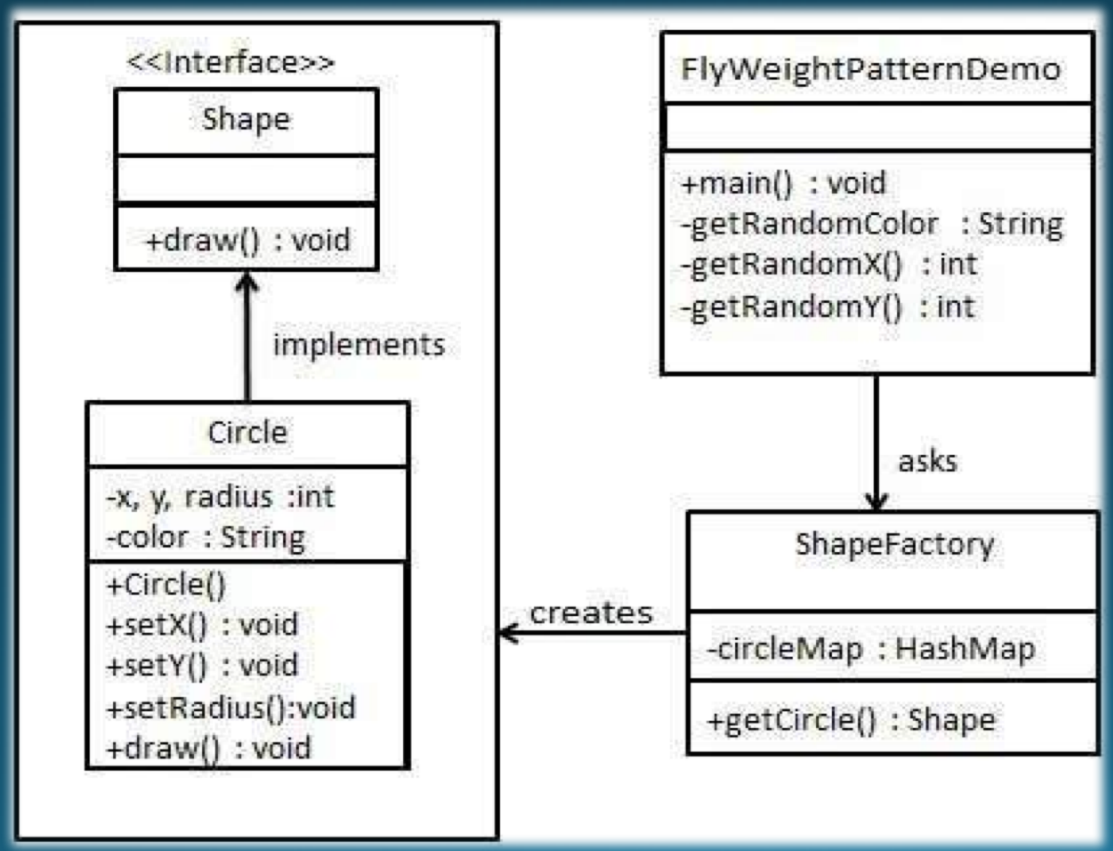
Structural patterns: Composite

- **Short Description:**



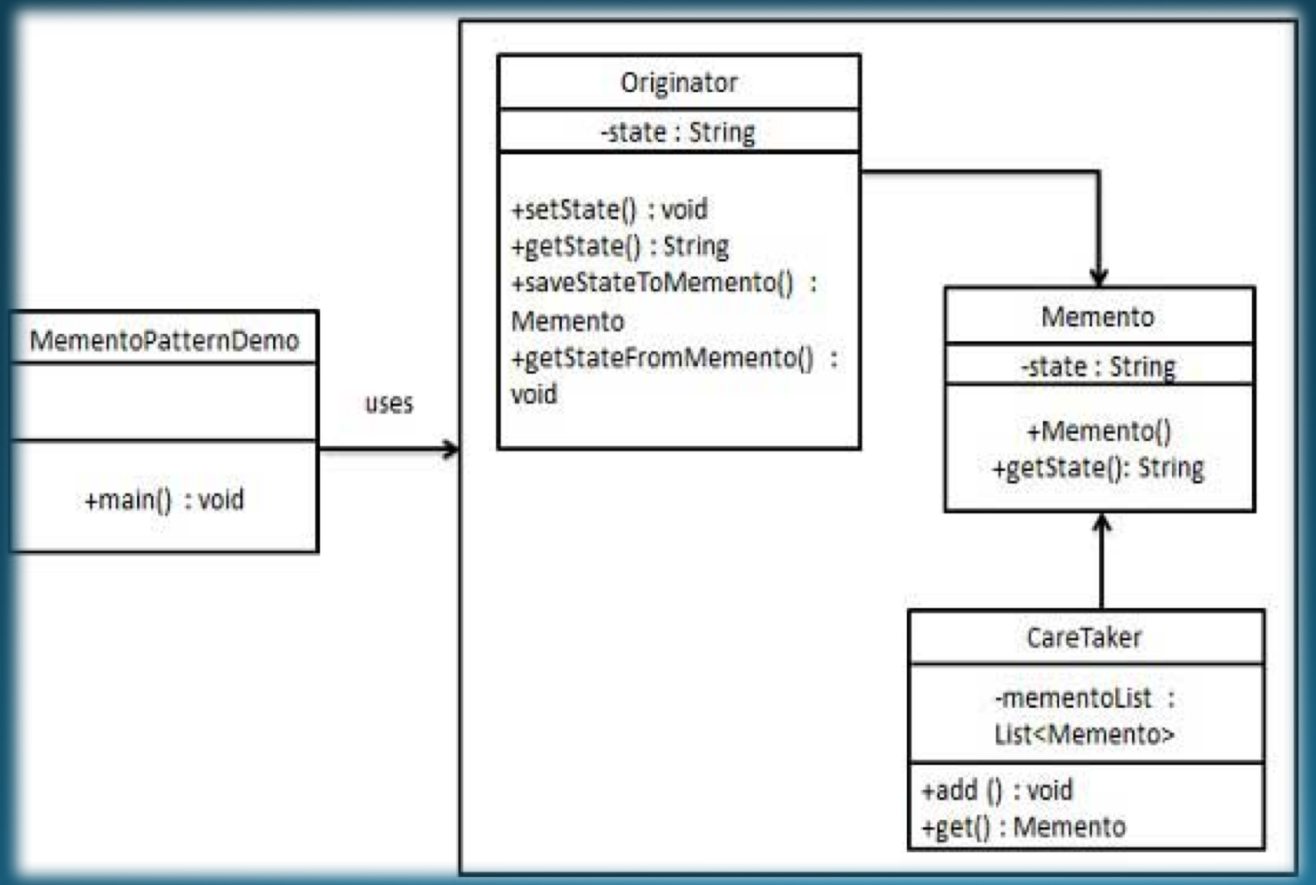
Structural patterns: Flyweight

- Short Description:



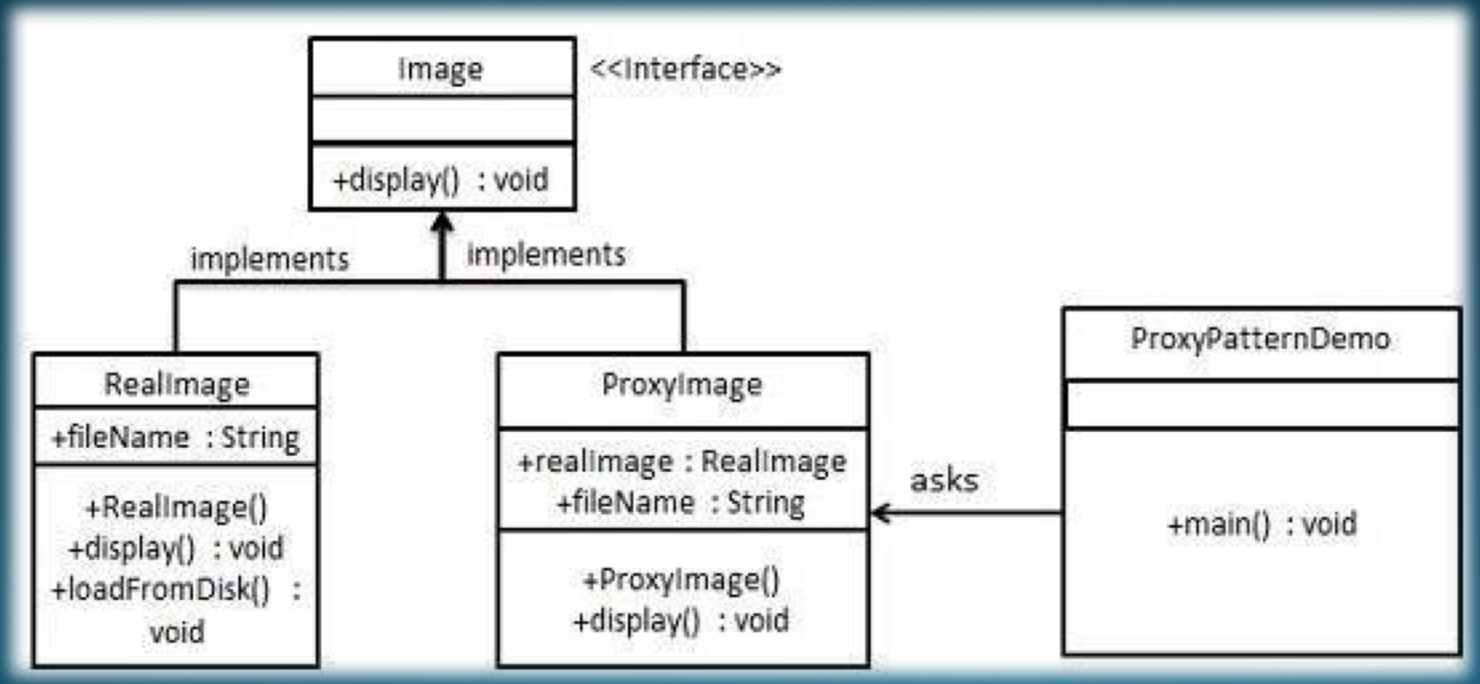
Structural patterns: Memento

- **Short Description:**



Structural patterns: Proxy

- **Short Description:**



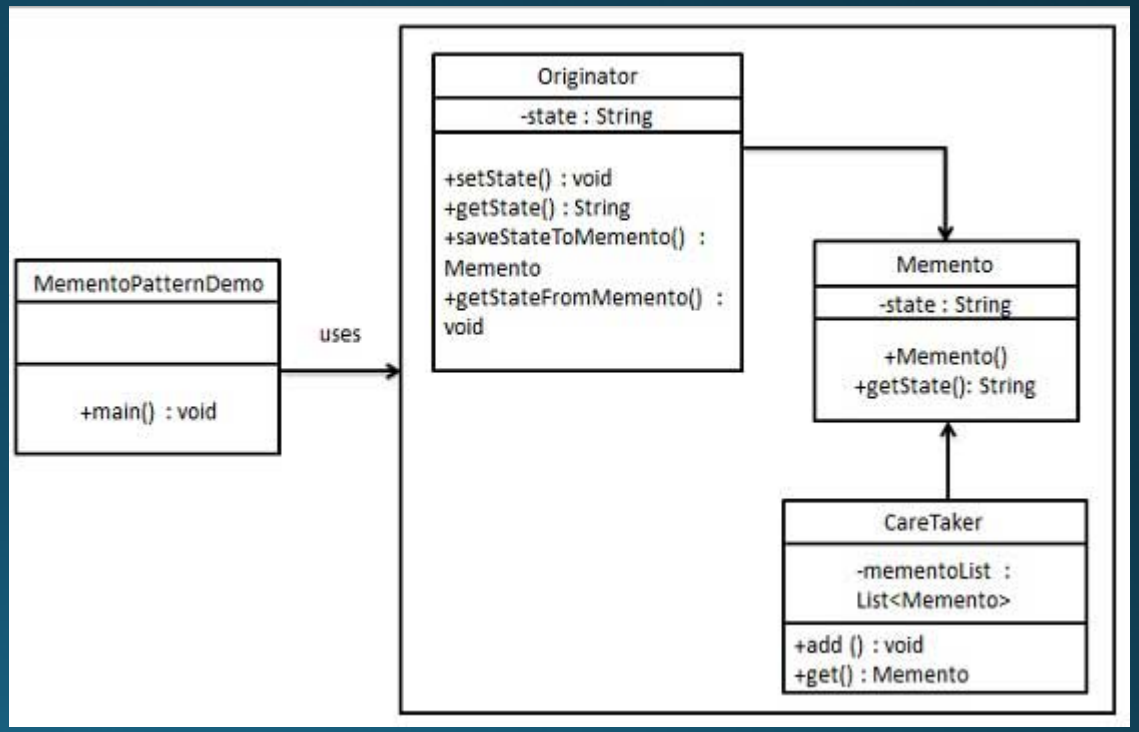
Behavoir patterns (GOF)

- Chain of Responsibility
- Command
- Interpreter
- **Memento**
- **Iterator**
- Mediator
- Observer
- State
- Strategy
- Template Method
- Visitor
- Null Object

“In software engineering, behavioral design patterns are design patterns that identify common communication patterns between objects and realize these patterns. By doing so, these patterns increase flexibility in carrying out this communication.”

Behavoir patterns (GOF): Memento

- Memento pattern tries to save different states of the class, as we can restore any state in any moment.
- It´s very useful when we have to modify the object but need to save its previous states, so we can get again the same object in their original state.



Behavoir patterns (GOF): Iterator

- This pattern is used to get a way to access the elements of a collection object in sequential manner without any need to know its underlying representation.

```
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);
   }
}
```

```
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;
      }
   }
}
```

Bibliography

https://sourcemaking.com/design_patterns
https://www.genbetadev.com/metodologias-de-programacion/patrones-de-diseno-que-son-y-por-que-debes-usarlos
https://msdn.microsoft.com/es-es/library/bb972240.aspx
https://www.fdi.ucm.es/profesor/jpavon/poo/2.14PDOO.pdf
http://c2.com/doc/oopsla87.html
http://www.buyya.com/254/Patterns/
https://en.wikipedia.org/wiki/Software_design_pattern
http://www.oodesign.com/
https://dzone.com/articles/design-patterns-abstract-factory
https://sourcemaking.com/design_patterns/builder/java/2
https://www.tutorialspoint.com/design_pattern/adapter_pattern.htm

Thanks for your attention