Software design patterns

Tarun Mohandas Daryanani
Abián Torres Torres

Content

- **Introduction**
  - Objectives
  - 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 designs allow to:

  - 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.”

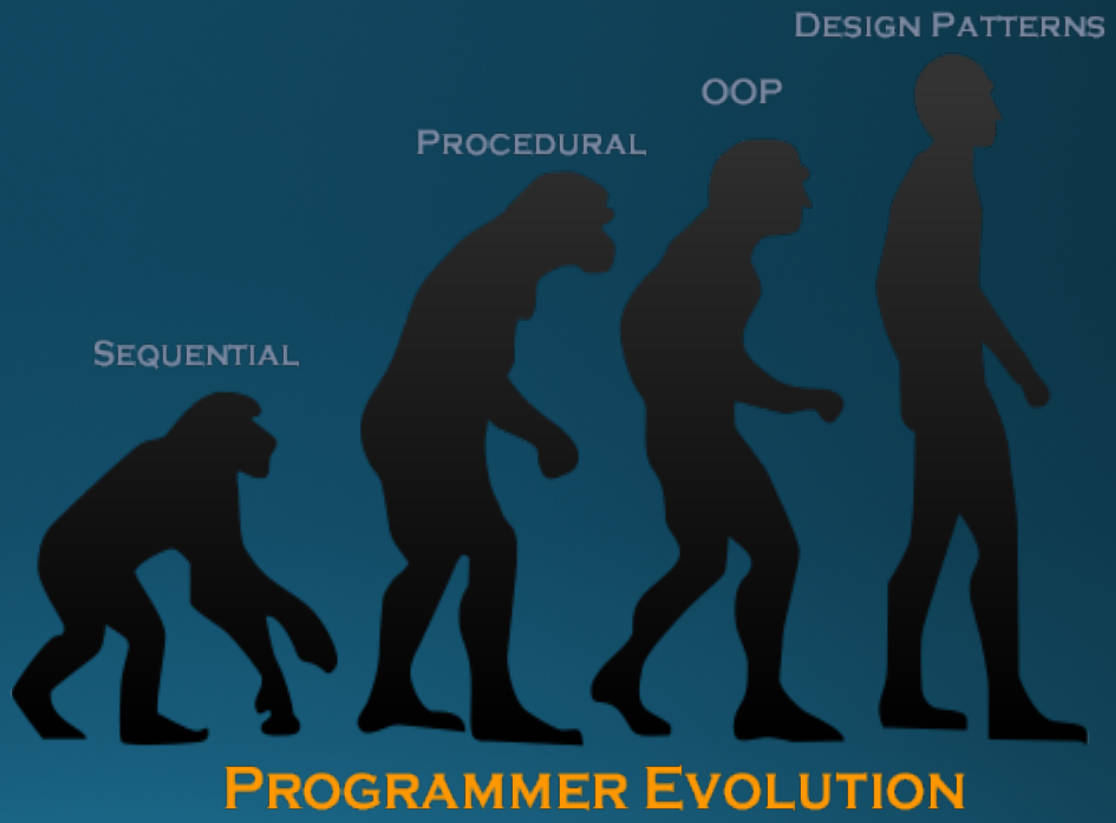
Pattern templates

- ***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*.**

oodesign
tutorialspoint

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

"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."

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.”

Behavoir patterns (GOF)

- **Chain of Responsibility**
* Command
* Interpreter
- **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."

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

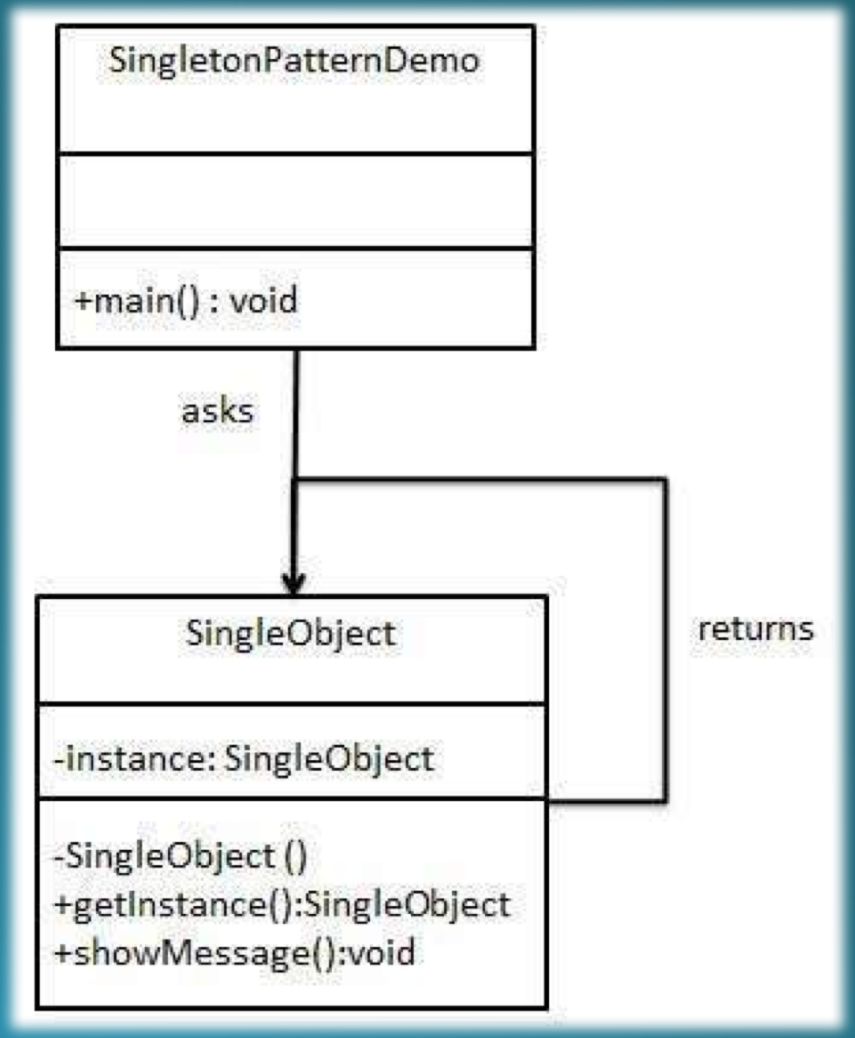
- **Applicability**:
  - According to the definition the singleton pattern should be used when there must be exactly one instance of a class, and when it must be accessible to clients from a global access point.

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

Creational patterns: Singleton

- **Implementation**

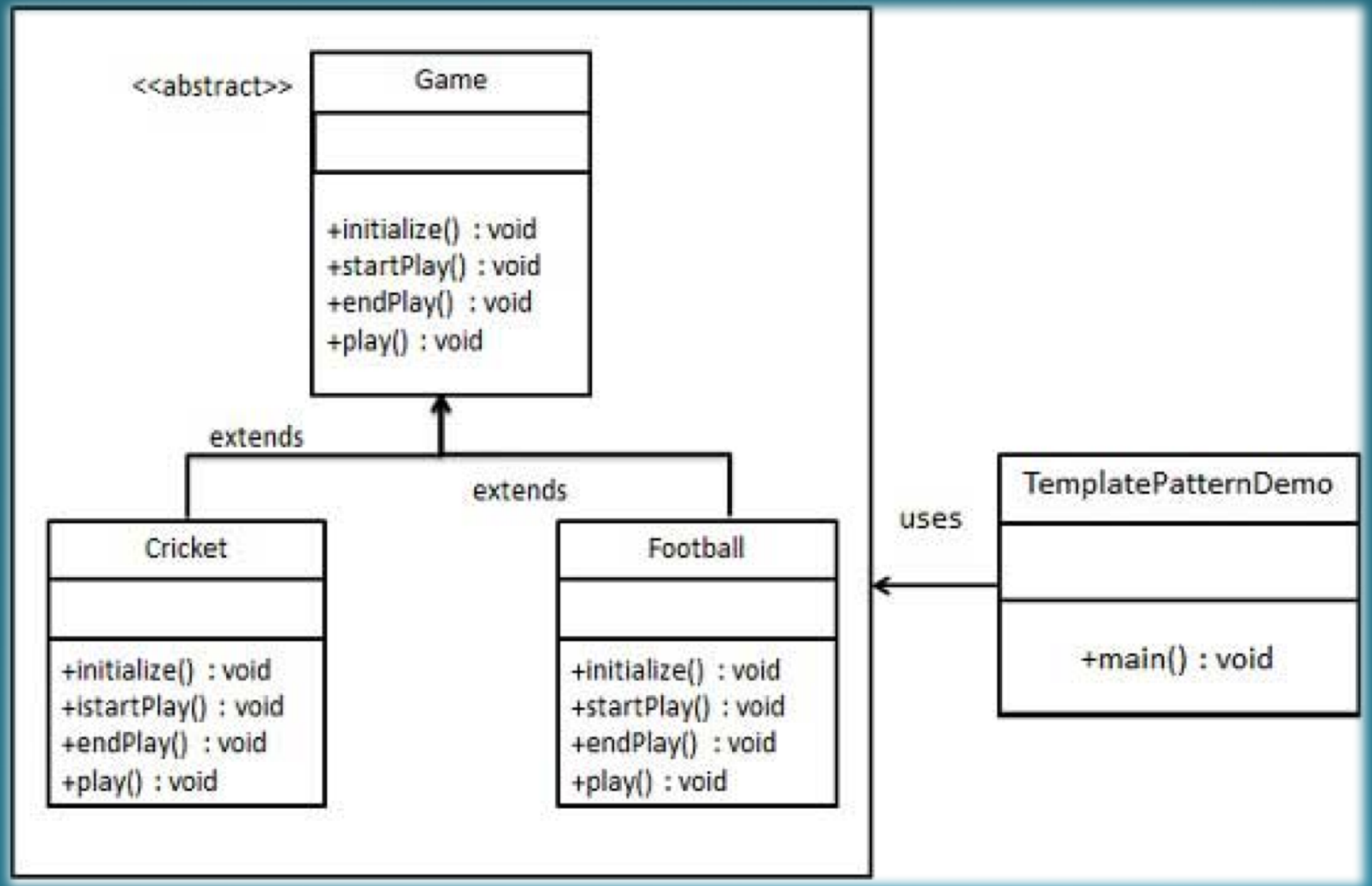
  The implementation involves a static member in the "Singleton" class, a private constructor and a static public method that returns a reference to the static member.



Behavioral patterns: Template Method

- Motivation:
  - A template method defines an algorithm in a base class using abstract operations that subclasses override to provide concrete behaviour.
  - So we are creating a template that we can concrete in more specific subclasses

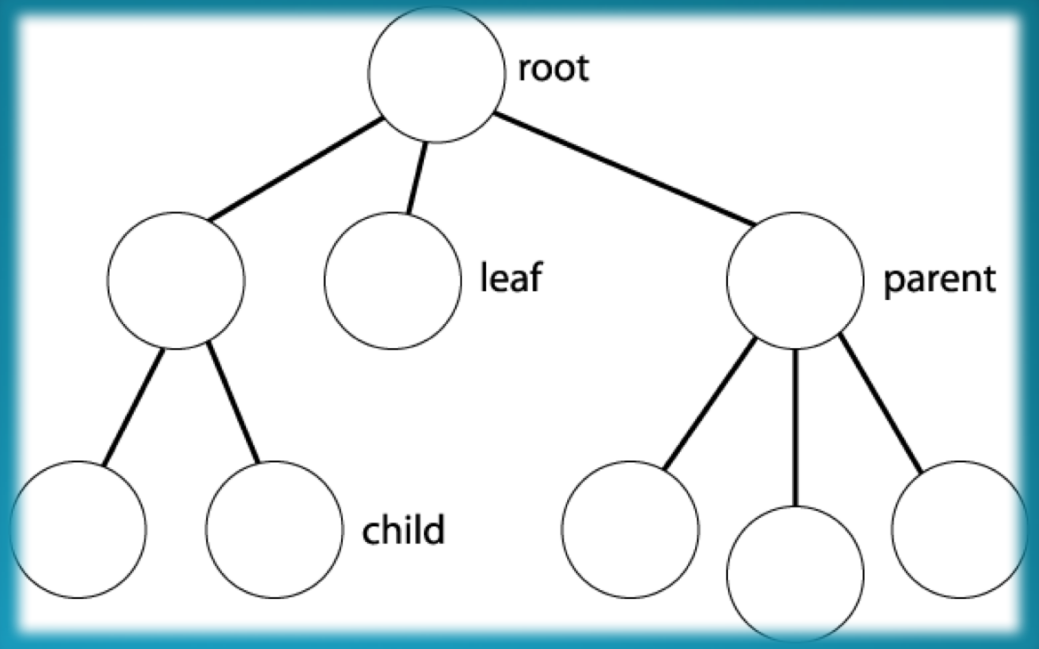
Behavioral patterns: Template Method



Structural patterns: Composite

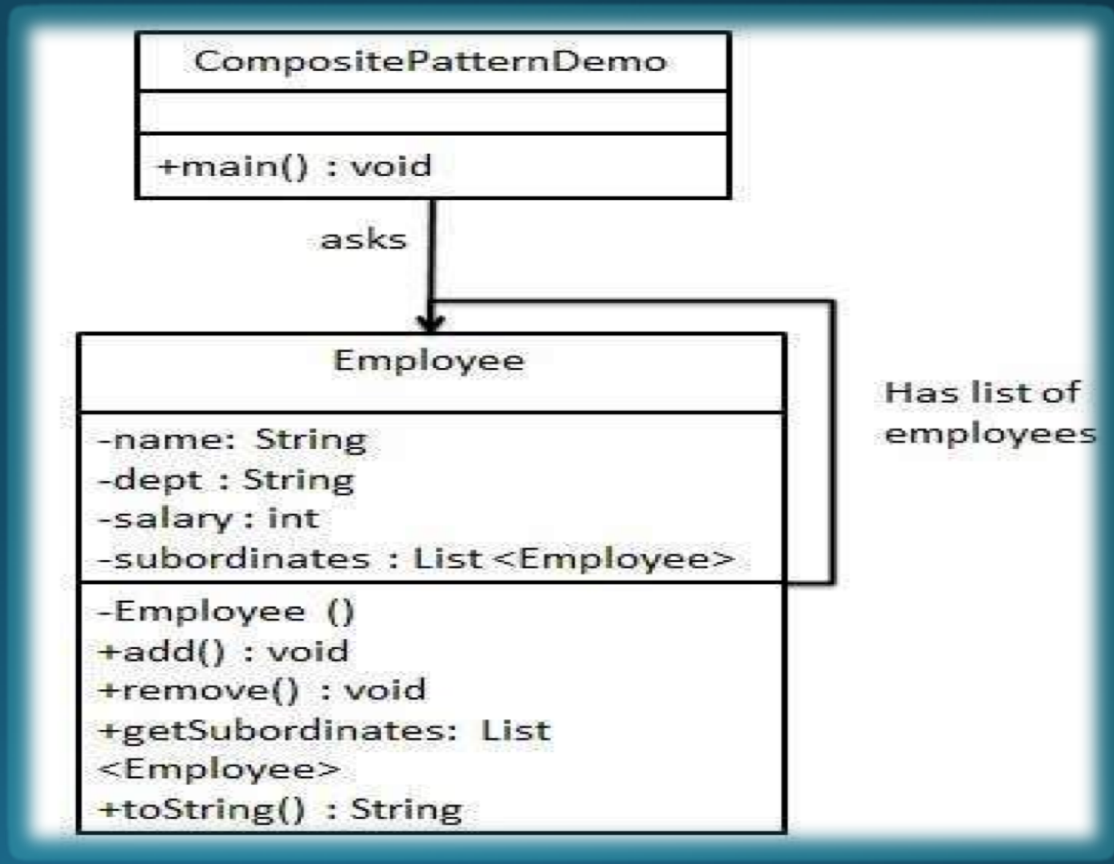
- Motivation:
  - There are times when a program needs to manipulate a tree data structure and it is necessary to treat both Branches as well as Leaf Nodes uniformly.

- Example: File System



Structural patterns: Composite

- **Implementation:**
  - Composite pattern is used where we need to treat a group of objects in similar way as a single object. Composite pattern composes objects in term of a tree structure to represent part as well as whole hierarchy.

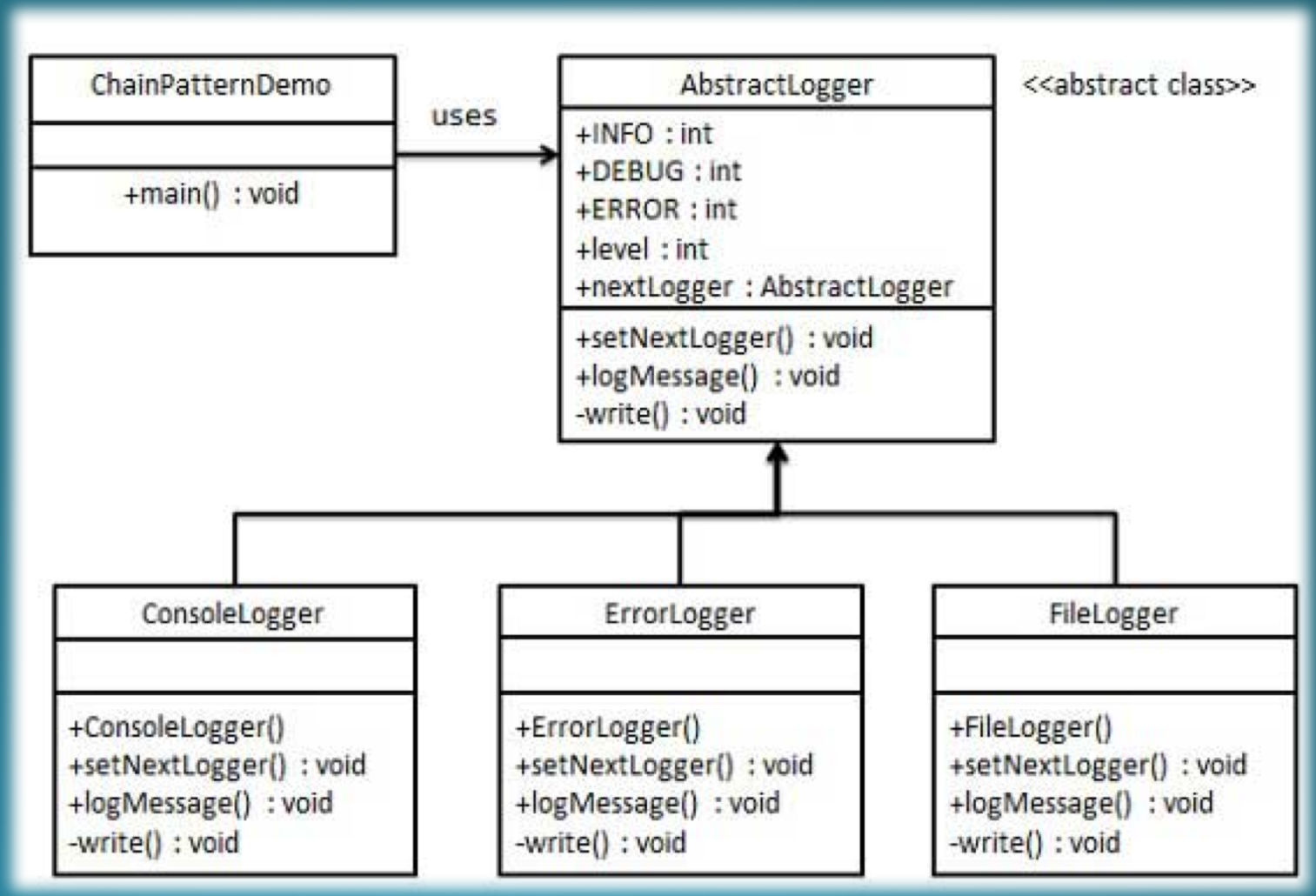


Behaviour patterns: Chain of Responsability

- Motivation:
  - In writing an application of any kind, it often happens that the event generated by one object needs to be handled by another one. And, to make our work even harder, we also happen to be denied access to the object which needs to handle the event

Behaviour patterns: Chain of Responsability

- Implementation:

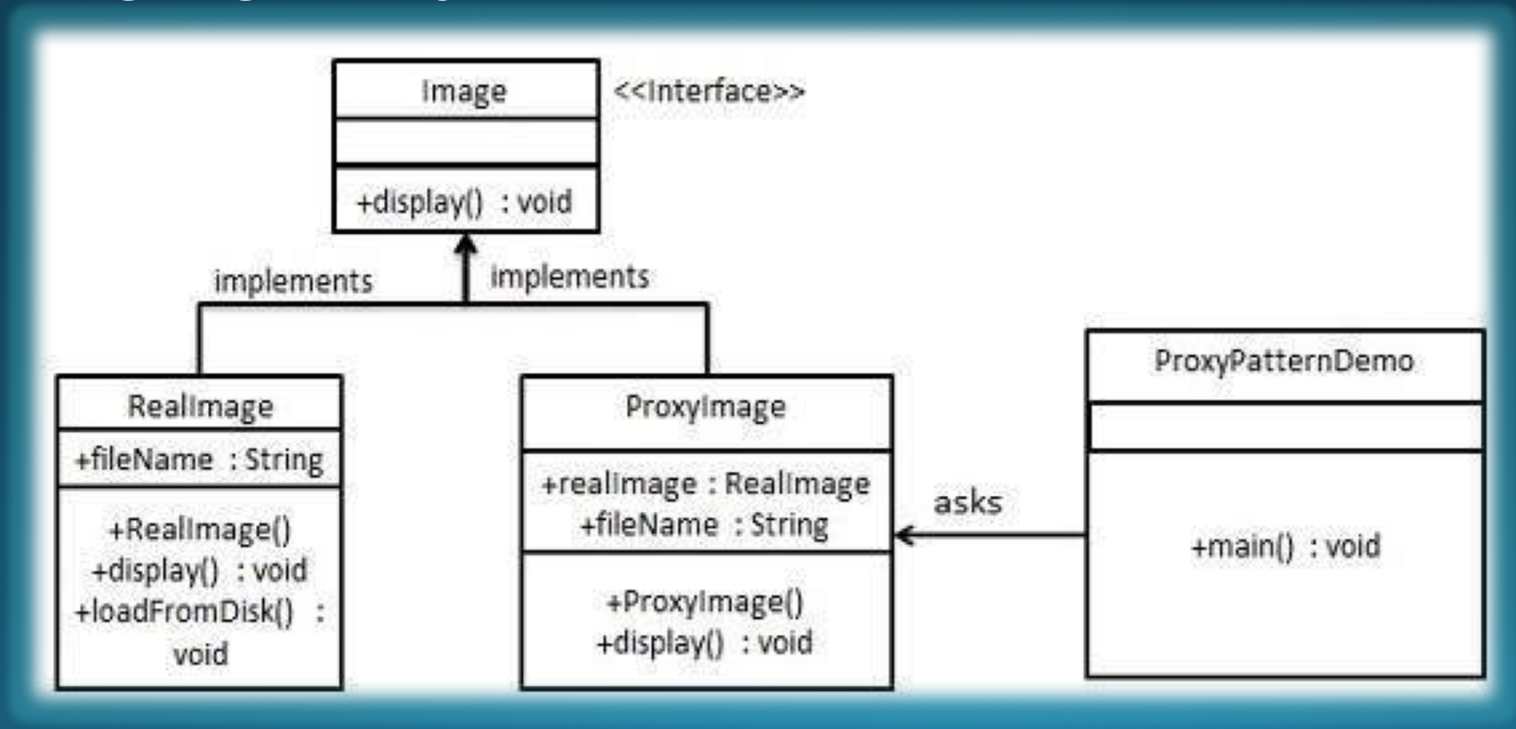


Behaviour patterns: Chain of Responsability

- Applicability:
  - When we want to handle an event by different objects in different ways
  - Also when we want to deny access by levels
- Examples:
  - Middlewares
  - ATM Machine

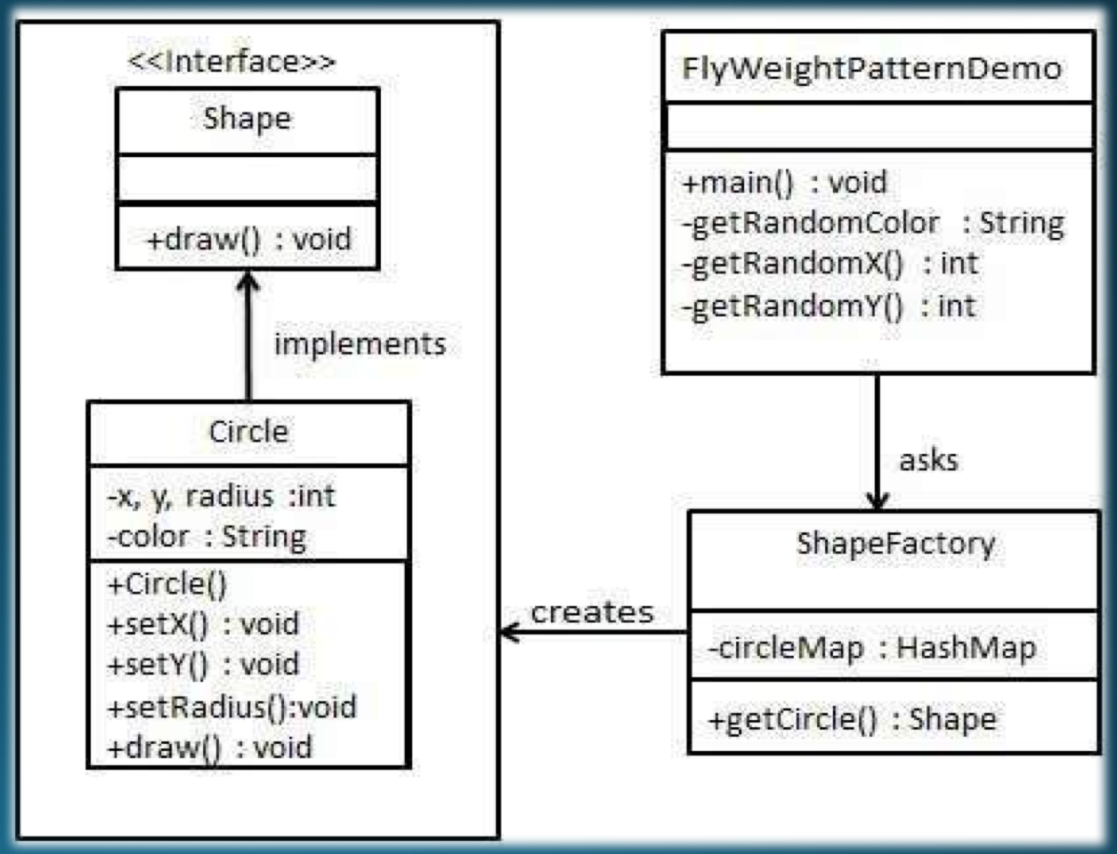
Structural patterns: Proxy

- **Short Description:**
  - In proxy pattern, a class represents functionality of another class. This type of design pattern comes under structural pattern. In proxy pattern, we create object having original object to interface its functionality to outer world.



Structural patterns: Flyweight

- **Short Description:**
  - Flyweight pattern is primarily used to reduce the number of objects created and to decrease memory footprint and increase performance.



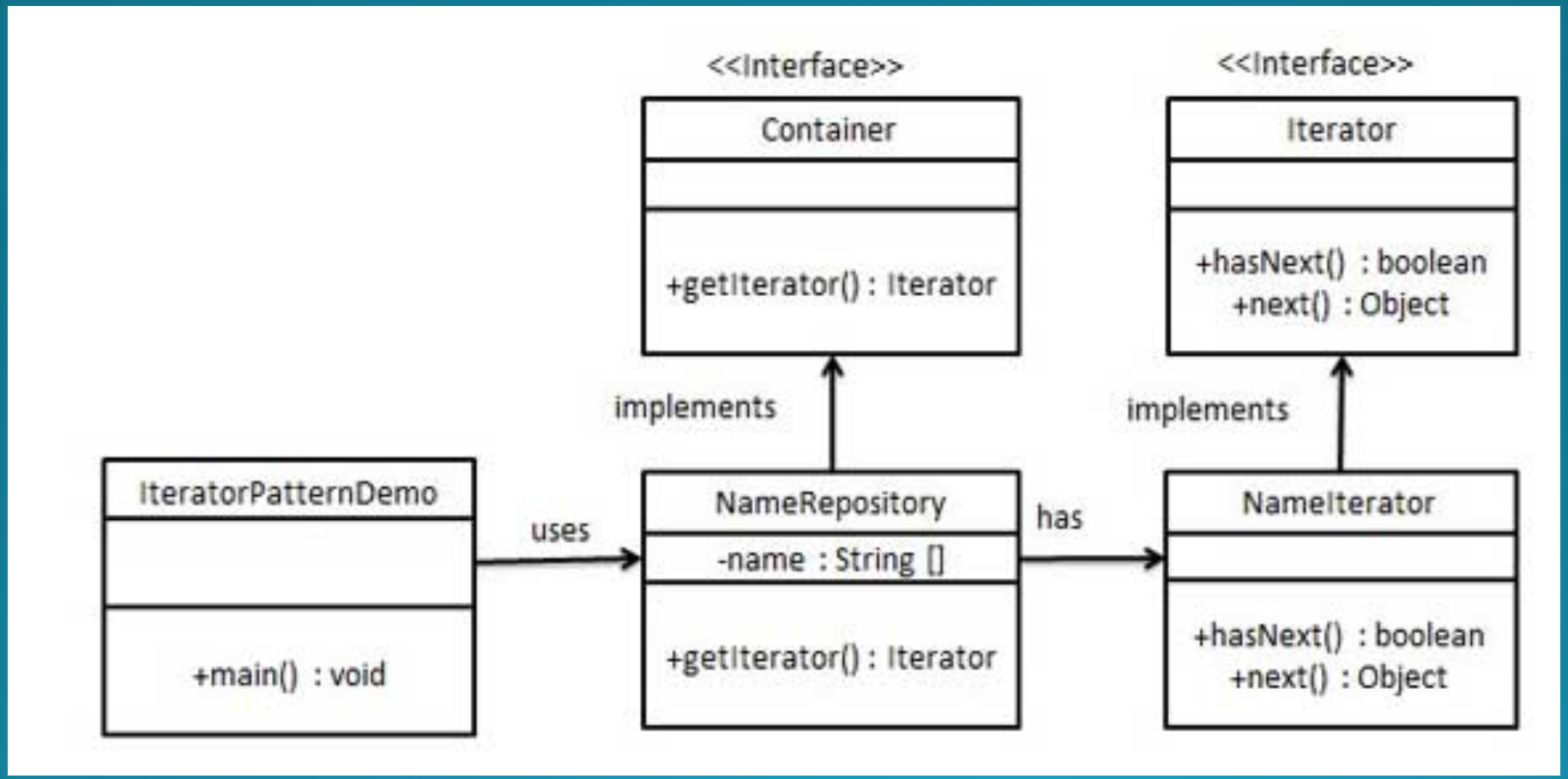
Behaviour patterns: Iterator

- Motivation:
  - One of the most common data structures in software development is what is generic called a collection. But what is more important is that a collection should provide a way to access its elements without exposing its internal structure.

  - The idea of the iterator pattern is to take the responsibility of accessing and passing through the objects of the collection and put it in the iterator object. This means abstracting us from the objects type.

Behaviour patterns: Iterator

- Implementation:



Behaviour patterns: Iterator

- Applicability:
  - The iterator pattern allow us to access contents of a collection without exposing its internal structure
  - Support multiple simultaneous traversals of a collection
  - Provide a uniform interface for traversing different collection.

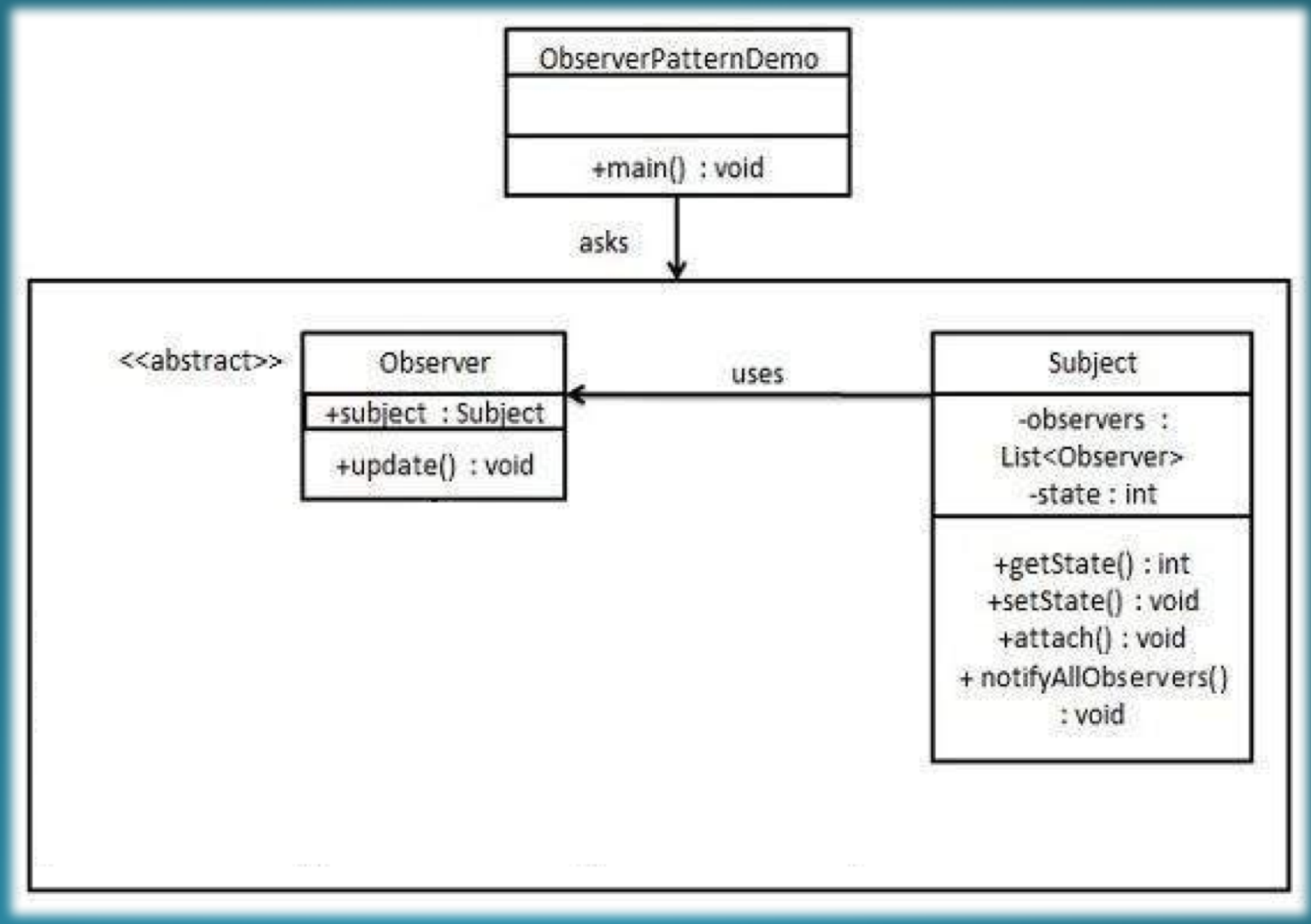
- Examples:
  - Scanner en java.
  - Set en c++.

Behavioral patterns: Observer

- Motivation
  - Observer pattern is used when there is one-to-many relationship between objects such as if one object is modified, its dependent objects are to be notified automatically. Observer pattern falls under behavioral pattern category.
  - Classes do not have to know about other classes' logic.

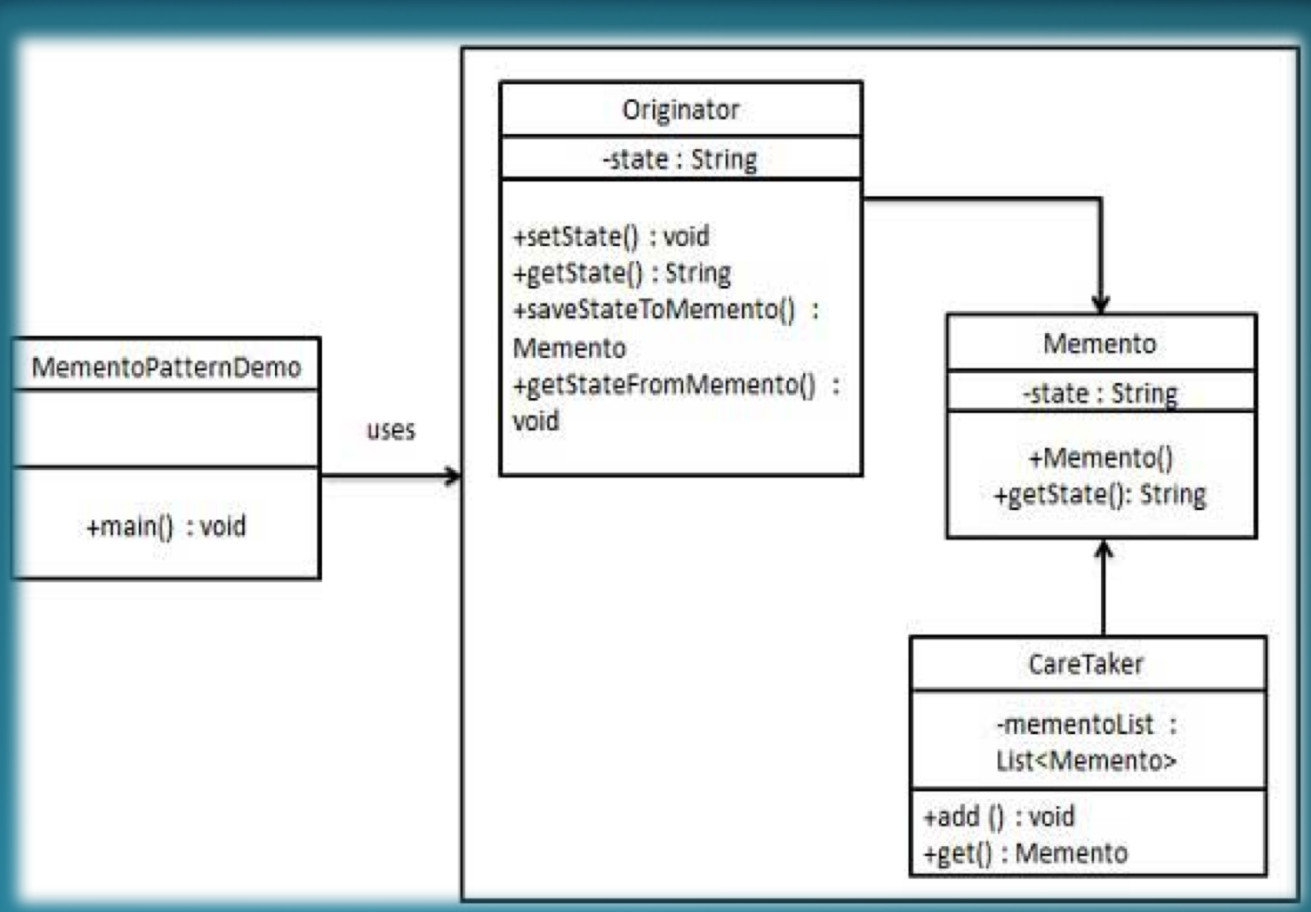
Behavioral patterns: Observer

- Implementation:



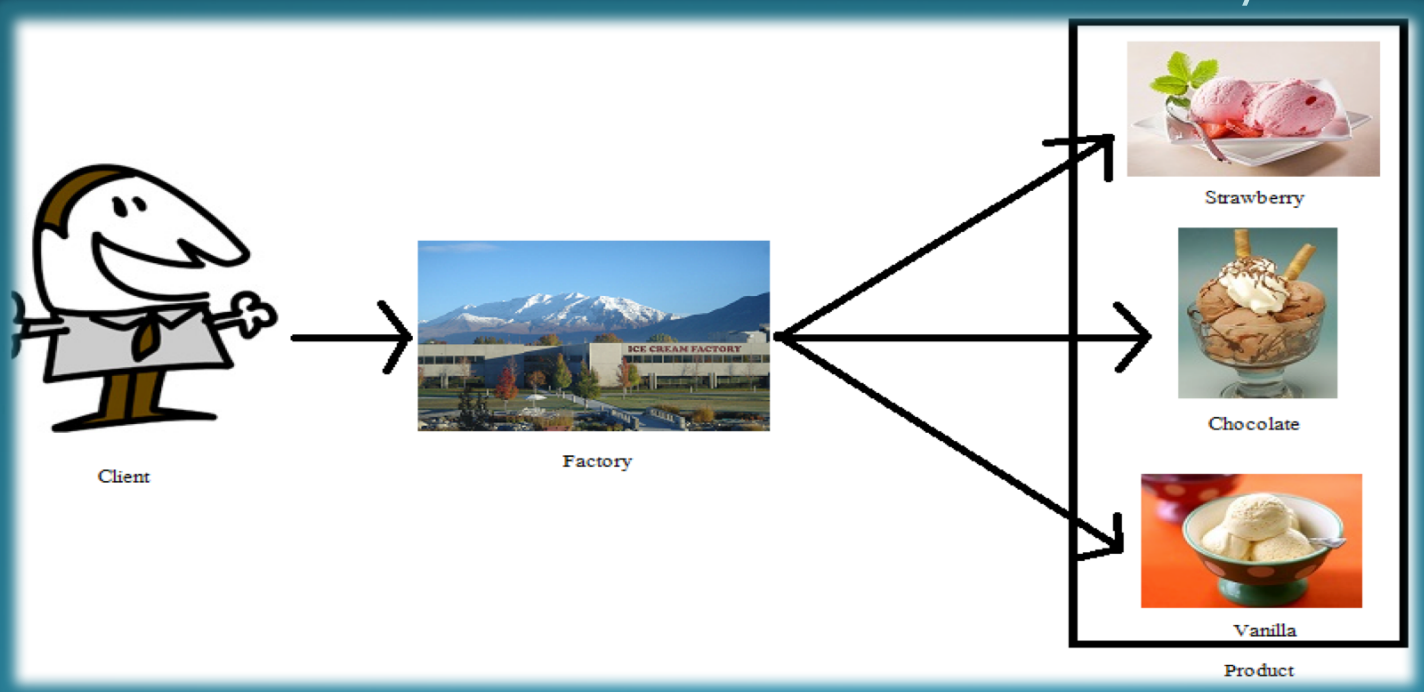
Structural patterns: Memento

- **Short Description:**
  - Memento pattern is used to restore state of an object to a previous state. Memento pattern falls under behavioral pattern category.



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.



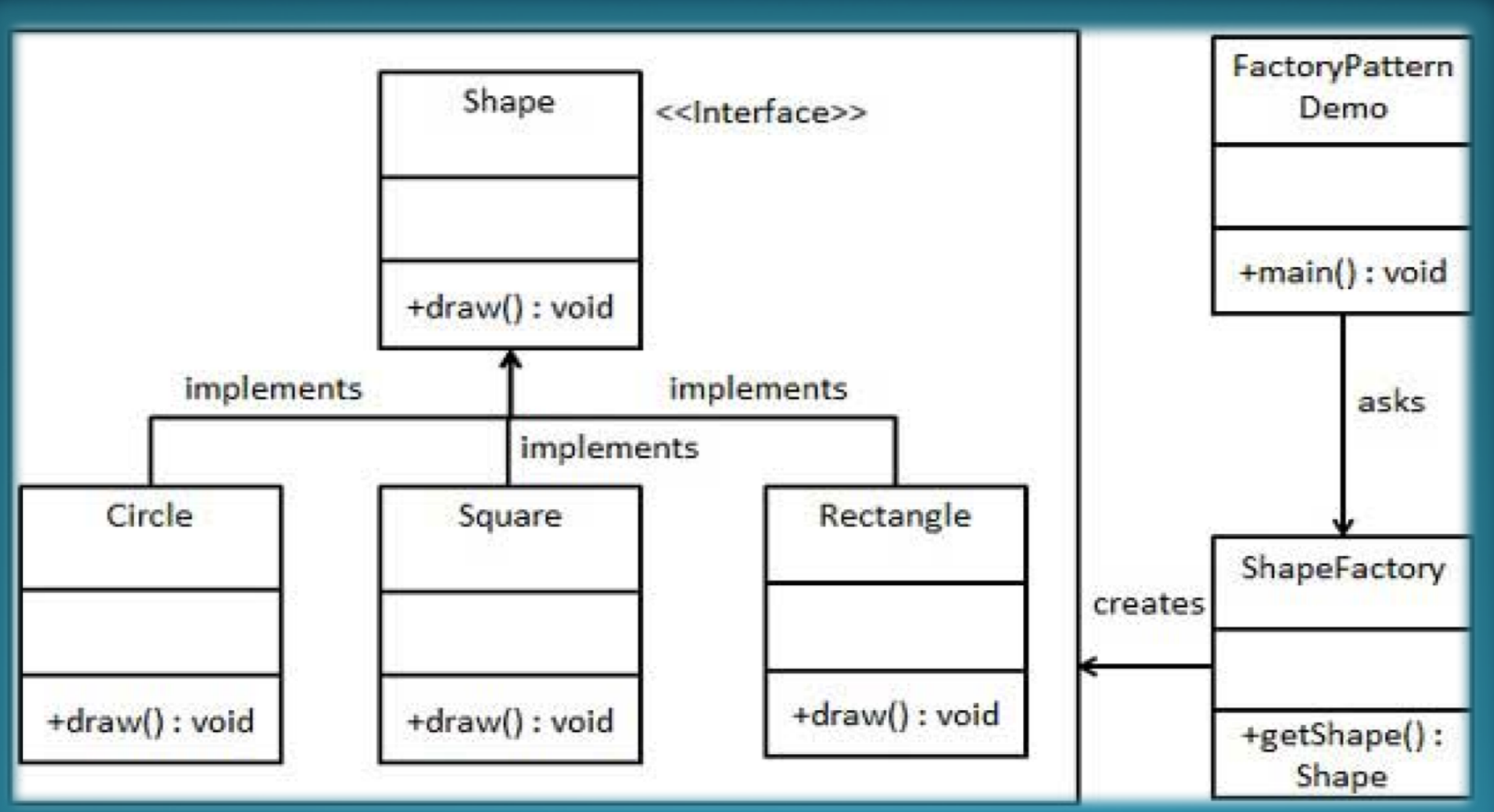
Creational patterns: Factory Method

- 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

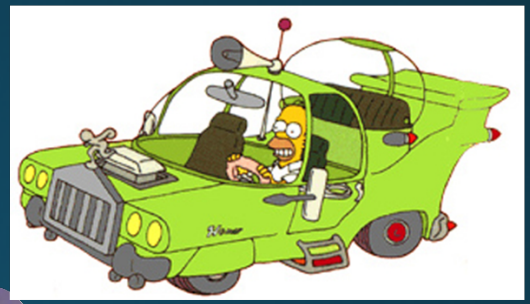
- Implementation:

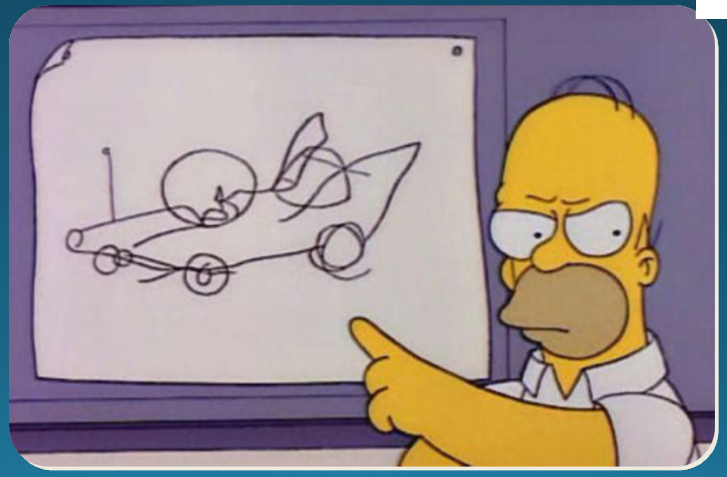


Creational patterns: Builder

- **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

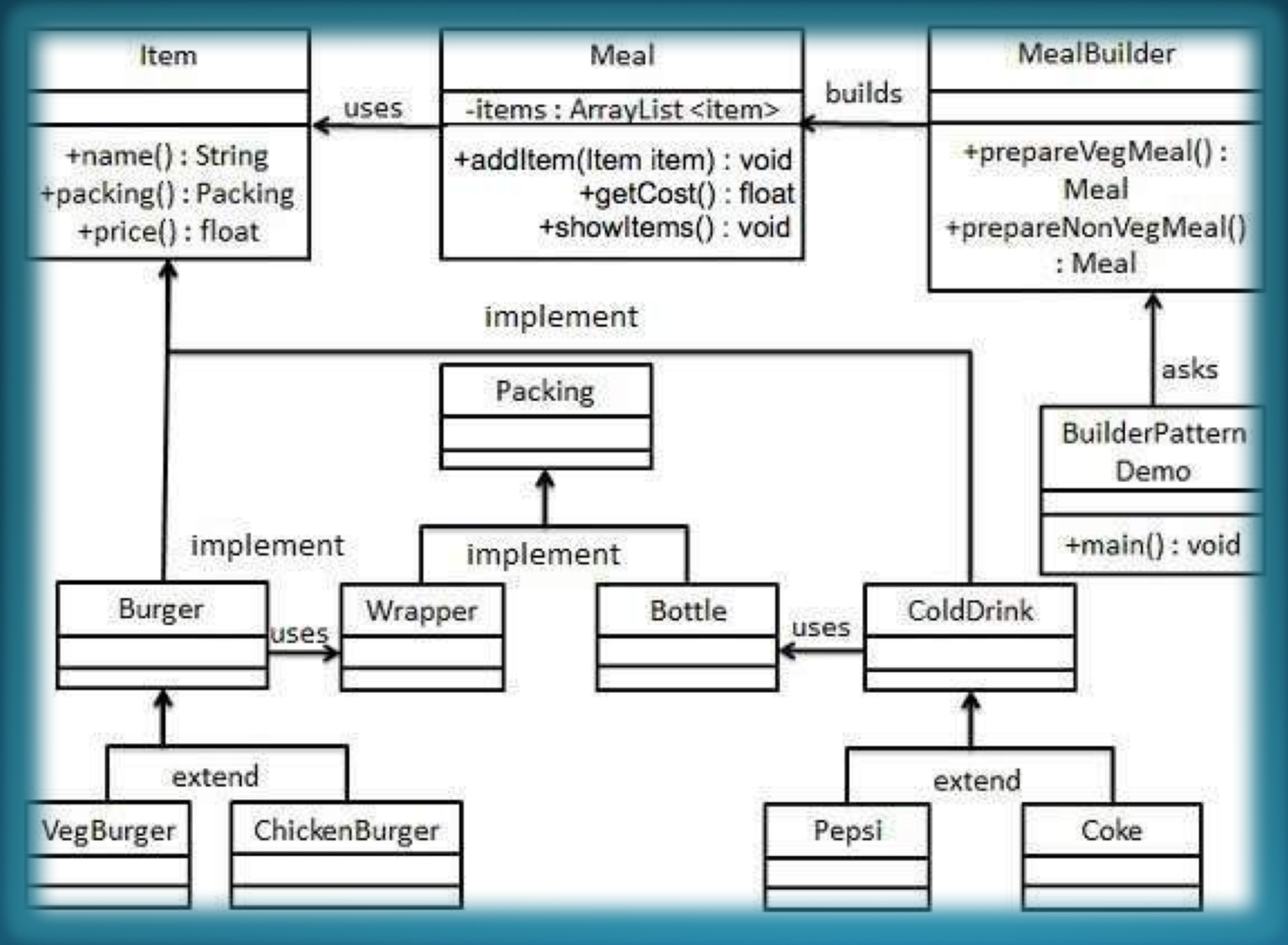




Emphasis on "what does it do?" more than "how it does?"

Creational patterns: Builder

- Example:



Structural patterns: Decorator

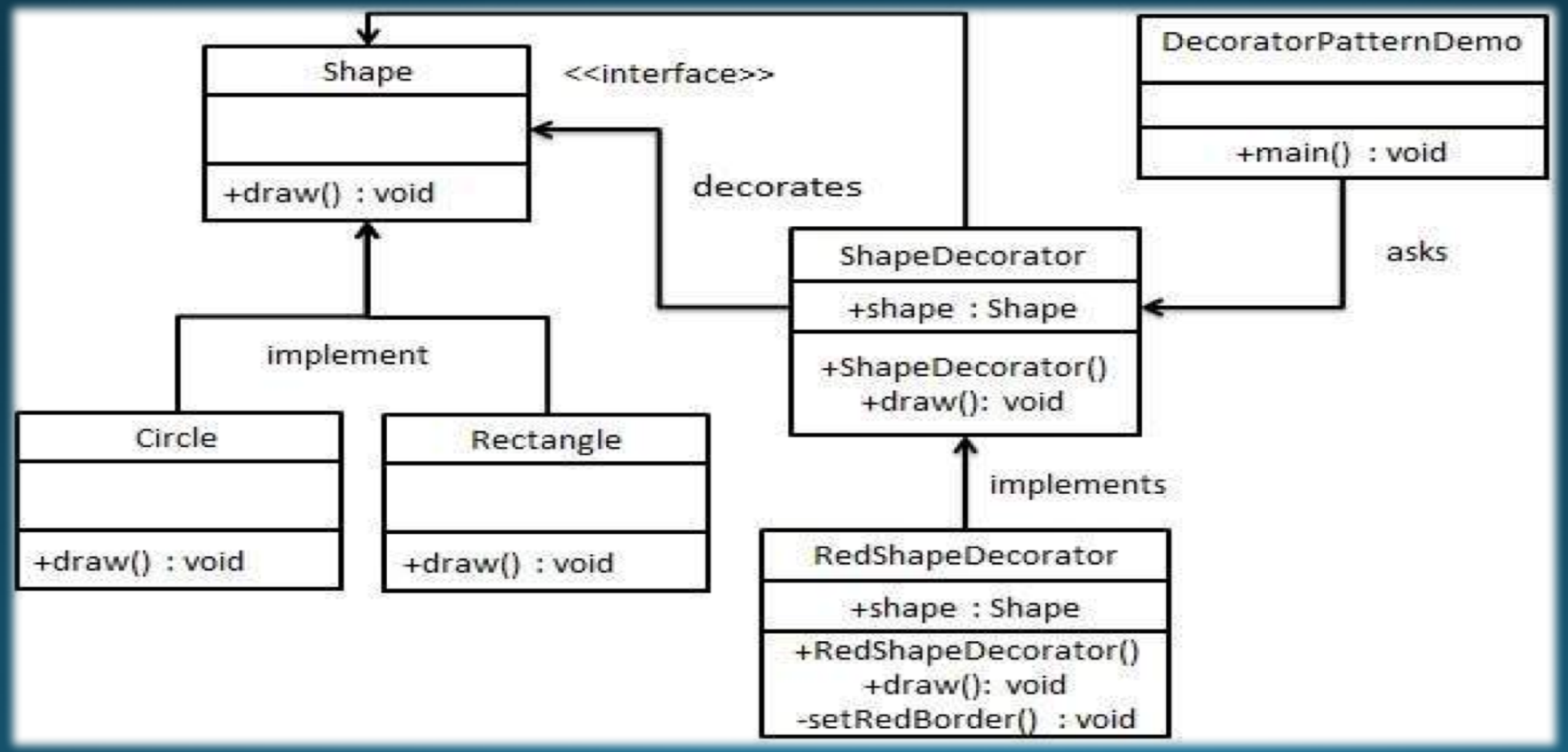
- Motivation
  - Decorator pattern allows a user to add new functionality to an existing object without altering its structure. This type of design pattern comes under structural pattern as this pattern acts as a wrapper to existing class.

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.

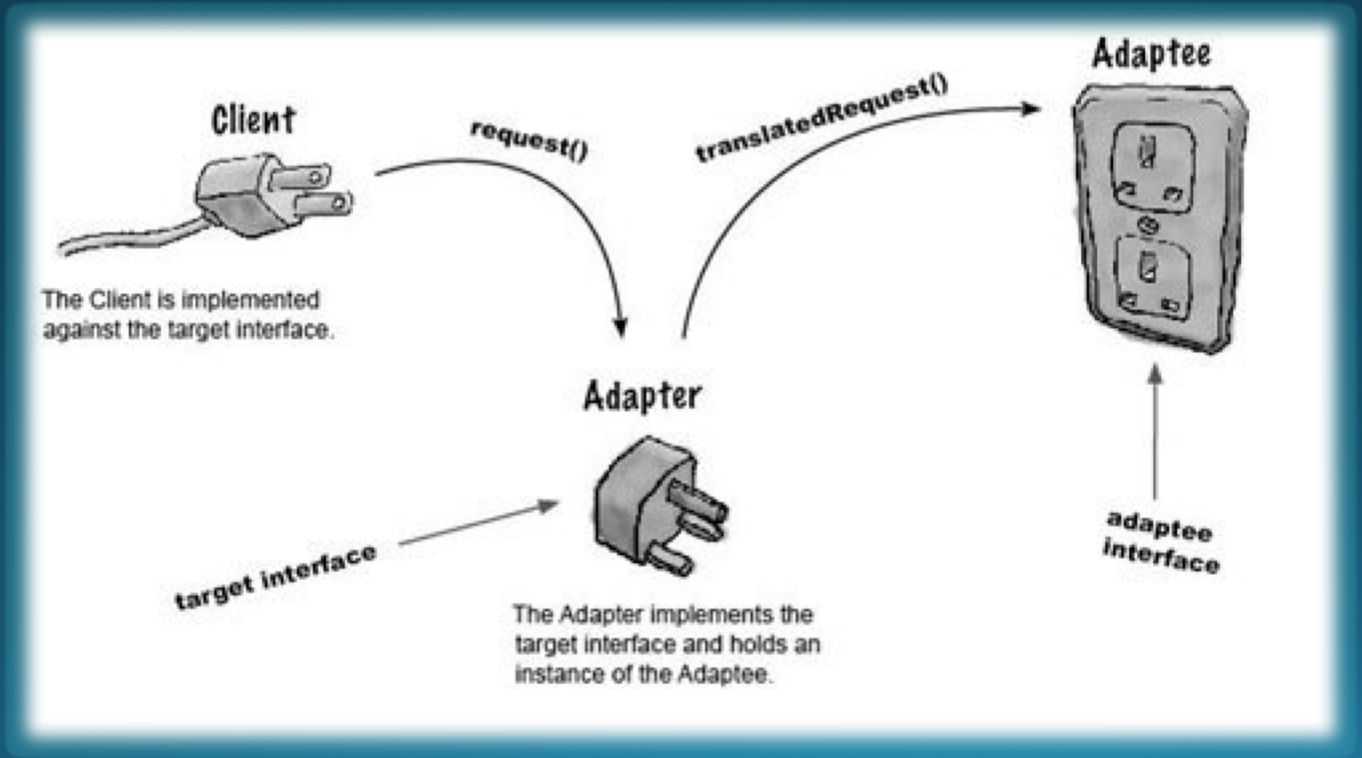
Structural patterns: Decorator

- Implementation



Structural patterns: 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.

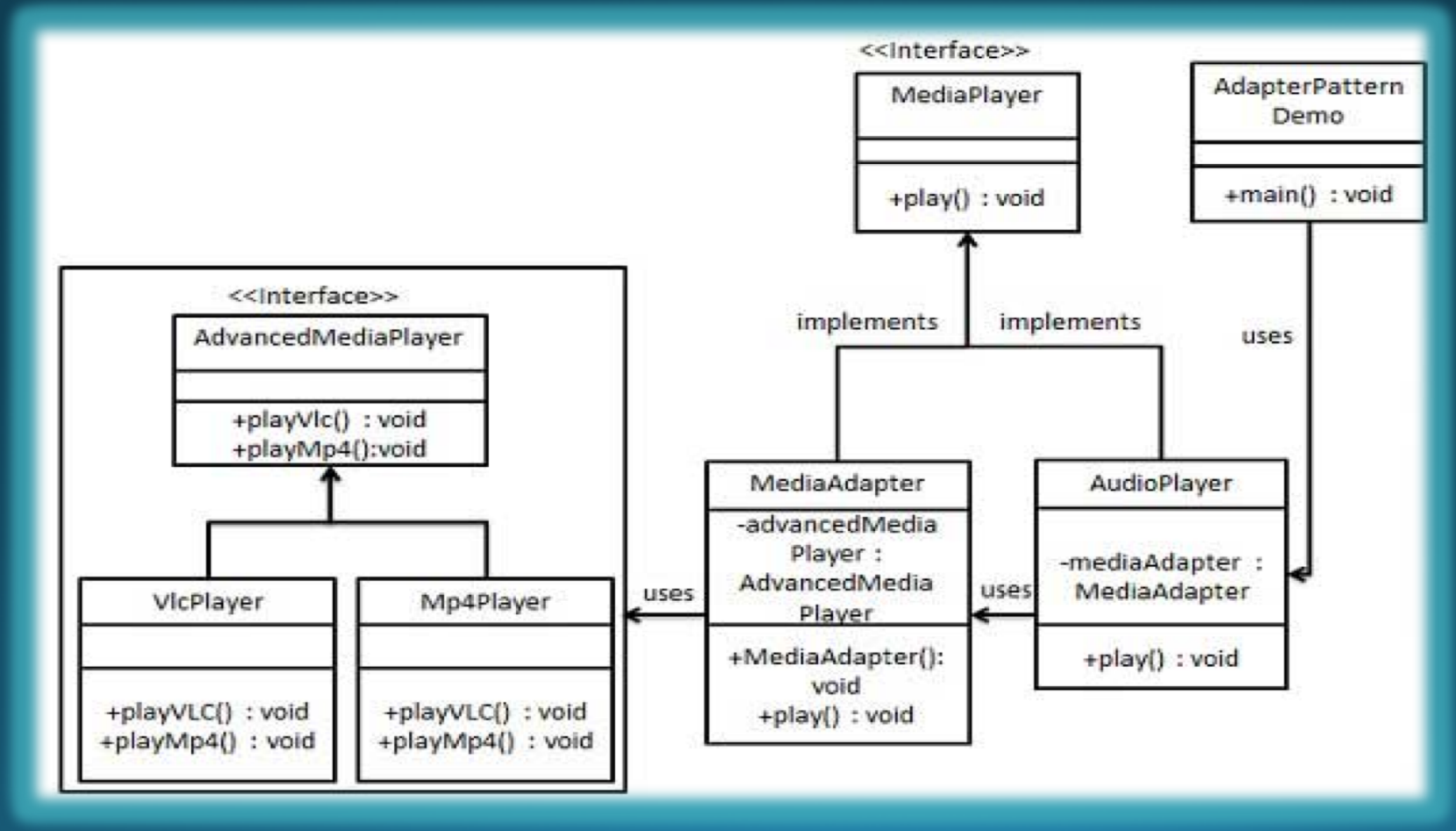


Structural patterns: Adapter

- **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:



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/

Thanks for your attention
Any Questions?