Java - Lesson 5

Design Patterns

Author: Kirill Volkov
https://github.com/vulko/JL5.DesignPatterns
vulkovk@gmail.com

Creational Patterns

- **Abstract factory**
  - Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- **Builder**
  - Separate the construction of a complex object from its representation, allowing the same construction process to create various representations.
- **Dependency Injection**
  - A class accepts the objects it requires from an injector instead of creating the objects directly.
- **Factory method**
  - Define an interface for creating a single object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- **Lazy initialization**
  - Tactic of delaying the creation of an object, the calculation of a value, or some other expensive process until the first time it is needed. This pattern appears in the GoF catalog as "virtual proxy", an implementation strategy for the Proxy pattern.
- **Multiton**
  - Ensure a class has only named instances, and provide a global point of access to them.
- **Object pool**
  - Avoid expensive acquisition and release of resources by recycling objects that are no longer in use. Can be considered a generalisation of connection pool and thread pool patterns.
- **Prototype**
  - Specify the kinds of objects to create using a prototypical instance, and create new objects from the 'skeleton' of an existing object, thus boosting performance and keeping memory footprints to a minimum.
- **Resource acquisition is initialization (RAII)**
  - Ensure that resources are properly released by tying them to the lifespan of suitable objects.
- **Singleton**
  - Ensure a class has only one instance, and provide a global point of access to it.

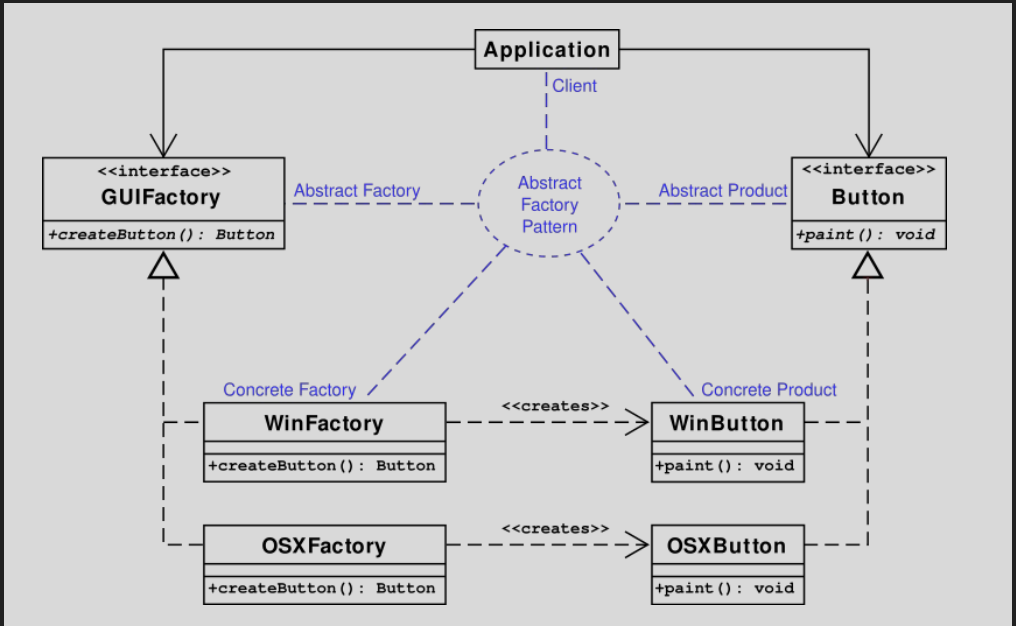
Abstract Factory

This pattern provides a way to encapsulate a group of individual factories that have a common theme without specifying their concrete classes.

Normally, client software creates a concrete implementation of an abstract factory and then uses generic interface of that factory to create objects.

Client doesn't care which concrete objects it gets from each of these internal factories, since it uses only generic interfaces of their products.

This pattern separates details of implementation of a set of objects from their general usage and relies on object composition.

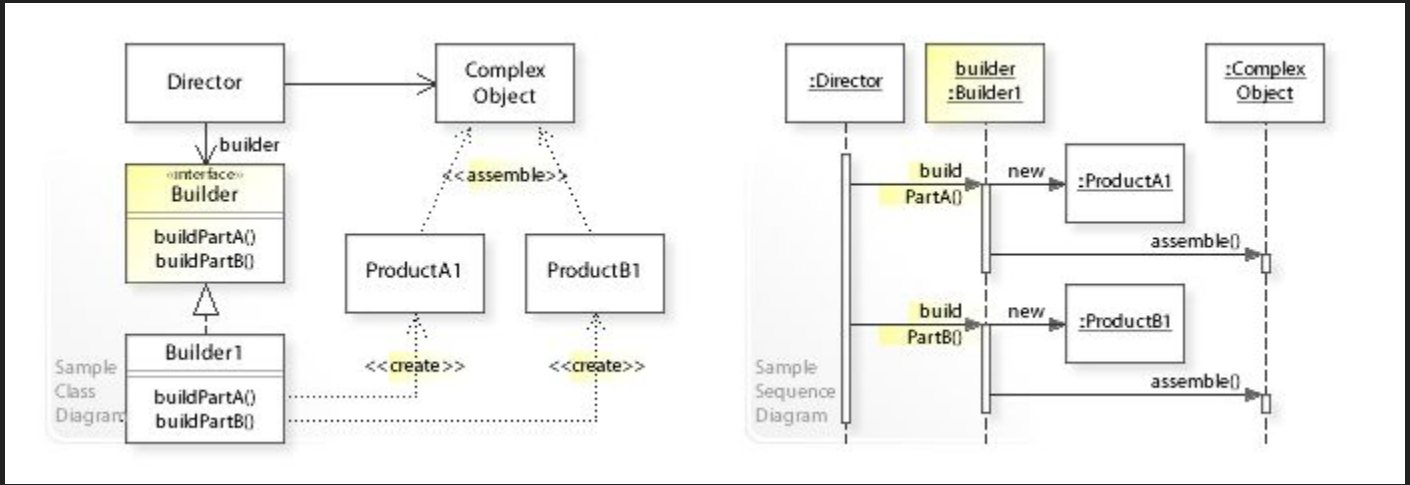


Builder

The Builder design pattern is a creational design pattern, designed to provide a flexible design solution to various object creation problems in Object-Oriented software.

The intent of the Builder design pattern is to separate the construction of a complex object from its representation.

Widely used in to create immutable objects.



Lazy init

```
public class Program {
    public static void main(String[] args) {
        Fruit.getFruitByTypeName(FruitType.banana);
        Fruit.showAll();
        Fruit.getFruitByTypeName(FruitType.apple);
        Fruit.showAll();
        Fruit.getFruitByTypeName(FruitType.banana);
        Fruit.showAll();
    }
}

enum FruitType { None, Apple, Banana }

class Fruit {
    private staticMap<FruitType, Fruit> types = new HashMap<>();
    private Fruit(FruitType type) {}

    public staticFruit getFruitByTypeName(FruitType type) { // Lazy factory method
        Fruit fruit;
        if (!types.containsKey(type)) {                    // Lazy initialisation
            fruit = newFruit(type);
            types.put(type, fruit);
        } else {
            fruit = types.get(type);                       // Already initialized
        }

        return fruit;
    }

    public static voidshowAll() {
        if (types.size() > 0) {
           System.out.println("Number of instances made = " + types.size());

           for (Entry<FruitType, Fruit> entry : types.entrySet()) {
               System.out.println(entry.getKey().toString());
           }

           System.out.println();
        }
    }
}
```

Singleton

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.

The concept is sometimes generalized to systems that operate more efficiently when only one object exists, or that restrict the instantiation to a certain number of objects.

Instance of a singleton can be acquired **wherever** and **whenever** it is needed.

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

    private Singleton() {}

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

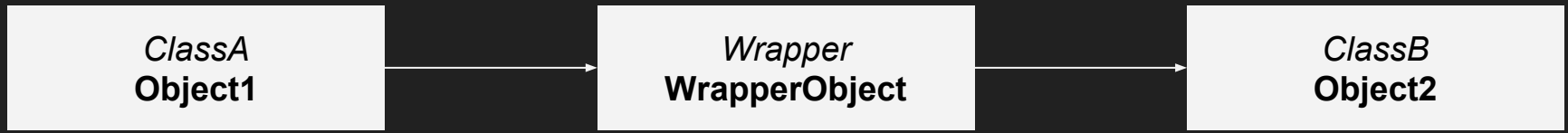
Structural Patterns

- Adapter, Wrapper, or Translator
  - Convert the interface of a class into another interface clients expect. An adapter lets classes work together that could not otherwise because of incompatible interfaces. The enterprise integration pattern equivalent is the translator.
- Bridge
  - Decouple an abstraction from its implementation allowing the two to vary independently.
- Composite
  - Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
- Decorator
  - Attach additional responsibilities to an object dynamically keeping the same interface. Decorators provide a flexible alternative to subclassing for extending functionality.
- Extension object
  - Adding functionality to a hierarchy without changing the hierarchy.
- Facade
  - Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface to makes it easier to use.
- Flyweight
  - Use sharing to support large numbers of similar objects efficiently.
- Front controller
  - The pattern relates to the design of Web applications. It provides a centralized entry point for handling requests.
- Marker
  - Empty interface to associate metadata with a class.
- Module
  - Group several related elements, such as classes, singletons, methods, globally used, into a single conceptual entity.
- Proxy
  - Provide a surrogate or placeholder for another object to control access to it.

Adapter and Wrapper

Adapter is a pattern that is used to make objects of different classes to access/modify each other without knowing anything about one another.

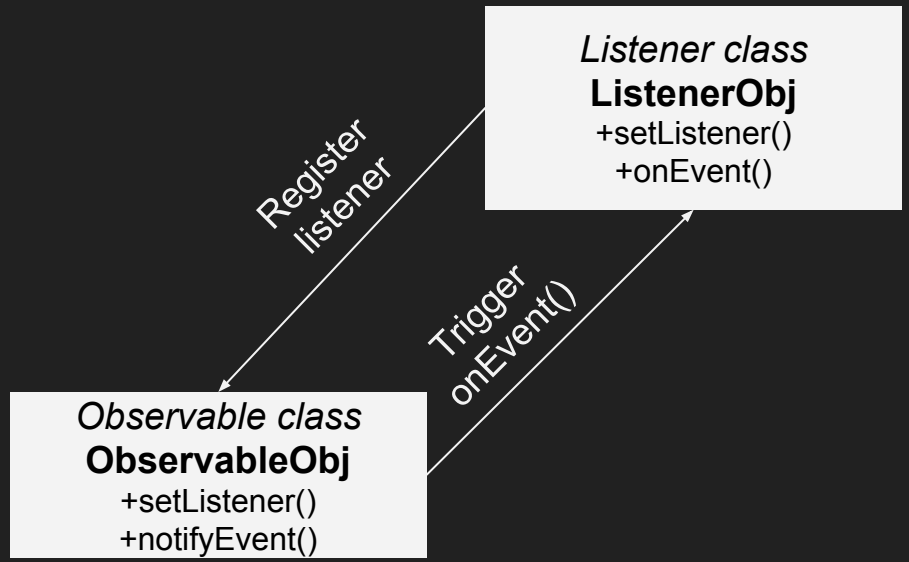
Wrapper is a pattern that allows another class to call methods of another class'es object without knowing anything about it.



Behavioral Patterns

- Chain of responsibility
  - Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
- Command
  - Encapsulate a request as an object, thereby allowing for the parameterization of clients with different requests, and the queuing or logging of requests. It also allows for the support of undoable operations.
- Iterator
  - Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- Null object
  - Avoid null references by providing a default object.
- Observer, Listener or Publish/Subscribe
  - Define a one-to-many dependency between objects where a state change in one object results in all its dependents being notified and updated automatically.
- Servant
  - Define common functionality for a group of classes.
- Specification
  - Recombinable business logic in a Boolean fashion.
- State
  - Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
- Strategy
  - Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- Template method
  - Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Listener pattern



State pattern