Design Patterns

by: Clyde Broderick

What We Will Discuss:

DESIGN PATTERNS

which showcase the best practices used by experienced object-oriented software developers.

CREATIONAL PATTERNS

1. They encapsulate knowledge about which concrete classes the system uses.
2. Hides how instances of classes are created and put together.
3. Eager vs. Lazy Initialization.

Singleton Pattern

One of the most simple design patterns in Java. One of the best ways to create an object!

Problem - Ensure that a class only has one instance and provide a global access point to it.

Example: ProfileManager profileManager = ProfileManager.getInstance(); //Singleton

Singleton allows you to design a class as if its use will be non-static (implementing interfaces, extending classes) but accessing it gives the appearance of static operations.

Solution: the instance of Singleton Class is created at the time of class loading.

Consequence: Instance is created even if client application might not be using it.

Factory Pattern

A method which is responsible for instantiating and returning an object.

Provides an interface for creating families of related or dependent objects without specifying their concrete classes.

Informal pattern in which there exists a class solely responsible for creation of another.

Builder Pattern

Builds a complex object using simple objects and using a step by step approach.

Problem: Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Resolves issues with large construction involving a lot of attributes.

Example: Consider an application wherein a License can consist of many different fields, often having null values.

Builder construction can be difficult to read and null values must be explicitly initialized.

STRUCTURAL PATTERNS

Eases design by identifying ways to realize relationships between entities while reducing code-redundancies.

- Facade Pattern
- Decorator Pattern
- Adapter Pattern
- Proxy Pattern

Facade Pattern

Provides a simplified interface to a larger body of code, such as a class library.

Prevents client from accessing additional methods that may lead to unexpected user-error.

LIMITS rather than EXTENDS functionality of an encapsulated object.

Consider the case where a client needs to keep a collection of items, but only needs to publicly expose 6 methods: add, remove, get, size(), foreach …

Decorator Pattern

Attach additional responsibilities to an object dynamically.

Provides a flexible alternative to subclassing for extending functionality.

Redundancies from this design are preventable by decorating Profile with a Player. Example being: public class Player extends Profile {

Adapter Pattern

Adapter pattern works as a bridge between two incompatible interfaces.

Converts interface of a class into another interface expected by client.

Allows classes to work together that couldn’t otherwise because of incompatible interfaces.

Example: Consider the case where a client needs to derive a Date from a LocalDate object.

Defining an Adapter Class:

Proxy Pattern

Proxy means ‘in place of’ or ‘on behalf of’.

Proxies are also called surrogates, handles, or wrappers. Avoids duplication.

A real world example is a check or credit card can be a proxy for our bank account. It can be used in place of our cash.

One of the advantages of proxies is its security.

- Remote Proxy
- Smart Proxy
- Virtual Proxy
- Protection Proxy

```java
package com.saket.demo.proxy;

import java.util.ArrayList;
import java.util.List;


public class ProxyInternet implements Internet
{
    private Internet internet = new RealInternet();
    private static List<String> bannedSites;

    static
    {
        bannedSites = new ArrayList<String>();
        bannedSites.add("abc.com");
        bannedSites.add("def.com");
        bannedSites.add("ijk.com");
        bannedSites.add("lnm.com");
    }

    @Override
    public void connectTo(String serverhost) throws Exception
    {
        if(bannedSites.contains(serverhost.toLowerCase()))
        {
            throw new Exception("Access Denied");
        }

        internet.connectTo(serverhost);
    }

}
```

BEHAVIORAL PATTERNS

- Characterizes the ways in which classes or objects interact and distribute responsibility.
- Concerned with algorithms and the assignment of responsibilities between objects.
- Describe object/class patterns as well as the resulting communication patterns between them.

Strategy Pattern

Concepts

- Eliminate conditional statements
- Behavior encapsulated in classes
- Difficult to add new strategies
- Client aware of strategies
- Client chooses strategy and knows different Strategies
- Class per Strategy

Observer Pattern

Concepts: One to Many Observers, Decoupled, Event Handling, Pub/Sub, M-V-C

Subject that needs to be observed.

Interface or Abstract class from which concrete implementation derive.

Observable - responsible for notifying registered observer's of state-change and updating.

Summary: Decoupled communication, built in functionality, used with mediator.

```
public interface Observable<T extends Observer> {
    void register(T observer);
    void unregister(T observer);
    void notify()
    List<Observer> getAllObservers();
}
```

Template Pattern

Create a method of high degree of freedom to define methods of lesser variability.

Degree of freedom is determined by the number of arguments of a method.

```
public static Integer [] getRange (inst start) {                    //degree of 1

public static Integer [] getRange (inst start, int stop) {          //degree of 2

public static Integer [] getRange (inst start, int stop, int step) {  //degree of 3
```

Command Pattern

Command Interface

- Decoupling “what is done” from “when it is done.”
- Concrete commands objects are tasks; i.e. - UNDO_TASK.
- “You’ll see command being used a lot when you need to have multiple undo operations.”
- To implement the undo, all you need to do is get the last Command in the stack.

Components of Command Pattern =

- Command Interface, Receiver Class, Invoker Class, Client Class