**DESIGN PATTERNS PAPER**

*Submitted by -: Revant Chandanala([rc296@njit.edu](mailto:rc296@njit.edu))*

Link to working project

<http://web.njit.edu/~rc296/IS683-Project1/Employees.html>

Link to github repository

**What are Design Patterns ?**

Design patterns are optimized, reusable solutions to the programming problems that we encounter every day while designing software. A design pattern is not a class or a library that we can simply plug into our system; it’s much more than that. It is a template that has to be implemented in the correct situation. Almost each language has it’s own Design Patterns and it is not language-specific. Design Patterns should be implemented carefully otherwise can create many problems if not implemented properly.

There are three basic categories of Design Patterns -:

a) **Structural Design Pattern**

b) **Behavioral Design Pattern**

c) **Creational Design Pattern**

**Structural** patterns generally deal with relationships between objects, making it easier for these objects to work together.

**Behavioral** patterns are used in communications between objects and make it easier and more flexible for these objects to communicate.

**Creational** patterns provide instantiation mechanisms, making it easier to create objects in a way that suits the situation.

**Why Should We Use Them ?**

Design patterns are well-thought out solutions to programming problems. Many programmers have encountered these problems before, and have used these solutions to remedy them. If we encounter such kind of problems, then why recreate a solution when we can use an already proven answer.

**Example**

Suppose that we have been given the responsibility of creating a way to merge two classes which do two different things based on the situation. These two classes are heavily used by the existing system in different places, making it difficult to remove these two classes and change the existing code. To add to this, changing the existing code requires that we also need to test any changed code in a system which relies on different components, which always has chance of introducing new bugs. Instead of doing this, we can implement a variation of various Design Patterns , which can easily handle these types of scenarios.

**Types Of Patterns**

1. **Strategy Pattern**

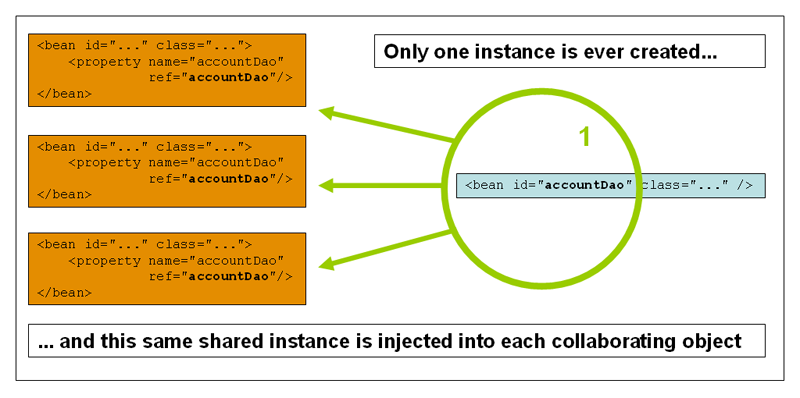


The **strategy pattern** is a behavioral design pattern that allows us to decide which course of action a program should take, based on a specific context during runtime.We can encapsulate two different algorithms inside two classes, and decide at runtime which startegy you want to go with.

**Where Can We Use It ?**

Suppose that we are currently developing a class which can either update or create a new user record. It still needs the same inputs (name, address, mobile number, etc.), but, depending on a given situation, it has to use different functions when updating and creating. Now, we could probably just use an if-else to accomplish this, however, what if we need to use this class in a different place? In that case, we will have to rewrite the same if-else statement all over again.

2. **Singleton Pattern**



The **singleton** design pattern is a creational design pattern which makes sure that you have one single instance of a particular class in the duration of your runtime, and provides a global point of access to the single instance.

**Where Can We Use It ?**

When used for namespacing and modularizing our code, the singleton pattern should be

used as often as possible. It is one of the most useful patterns in JavaScript and has its place in

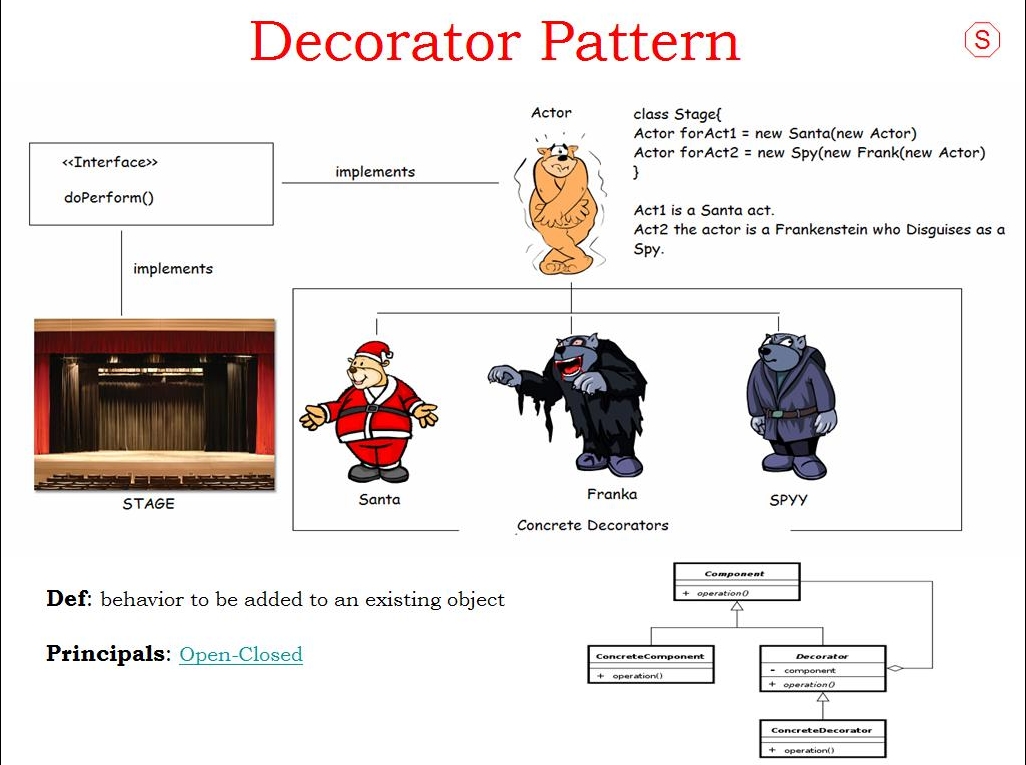
almost every project, no matter how large or small. In quick and simple projects, a singleton

can be used simply as a namespace to contain all of our code under a single global variable.

On larger, more complex projects, it can be used to group related code together for easier

maintainability later on, or to house data or code in a single well-known location.

3. **Decorator Pattern**



The **decorator** pattern is a structural design pattern which enables us to add new or additional behavior to an object during runtime, depending on the situation.

**Where Can We Use It ?**

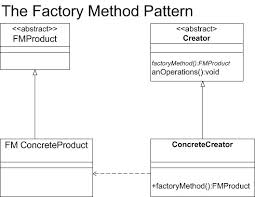
The decorator pattern should be used when you want to add features to an object

without having to change the code that uses it. Since decorators can modify objects dynamically

and transparently, they are perfect for modifying existing systems. It can often be easier

to create and apply a few decorators than it is to go through the trouble of creating and maintaining a subclass.

4. **Factory Method Pattern**



The **factory method** pattern is a creational design pattern which does exactly as it sounds: it’s a class that acts as a factory of object instances.

**Where Can We Use It ?**

Suppose we have an online bookstore and people come to our website to buy books. When a customer orders a book, we just have another book distributor send the book directly to the customer. We are a middle man and we don’t stock the books. We have many distributors that we can choose to send the books directly to our customers. The key is our customer should not care which distributor we choose because they will get their books regardless. It is completely hidden from the customer's point of view, and they should not be concerned about it. We, the online bookstore, are the ones that determines the distributor to use.

5. **Adapter Pattern**

The **adapter pattern** is a structural design pattern that allows you to repurpose a class with a different interface, allowing it to be used by a system which uses different calling methods.

**Where Can We Use It ?**

Adapters should be used in any place where clients expect a particular interface but the interface

offered by the existing API is incompatible. Adapters should only be used to reconcile

differences in syntax; the method you are adapting still needs to be able to perform the needed

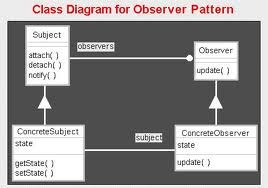
task. If this is not true, an adapter will not solve your problem. Adapters can also be used when

clients prefer a different interface, perhaps one that is easier for them to use. When you create

an adapter, just like a bridge or a facade, you decouple an abstraction from its implementation,

allowing them to vary independently.

6. **Observer Pattern**



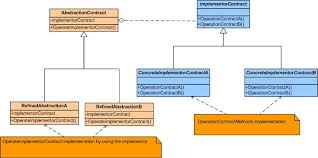
In the Observer Pattern, a subject can have a list of observers that are interested in it’s lifecycle. Anytime the subject does something interesting, it sends a notification to its observers. If an observer is no longer interested in listening to the subject, the subject can remove it from its list.

**Where We Can Use It ?**

The observer pattern should be used in situations where you want to abstract human behavior

from application behavior. It’s best not to implement something that is tied to user interaction and originates from the browser, such as basic DOM events like click, mouseover, or keypress. None of these events are useful pieces of information to an implementer who simply wants to know when an animation begins, or when a word is spelled incorrectly in a spell-check application.Let’s say for example that when a user clicks a tab in a navigation system, a menu with more information about the tab is toggled. Granted, you could simply just listen for the click event, but this requires knowing which element to listen for. There is another downside: you’ve now tied your implementation directly to the click event. Instead of listening for the click event, it would be better to simply create an onTabChange observable object and allow observers to be notified when the particular event occurs. Since these tabs could in fact be toggled on mouseover or even on focus, this is something that the observable object would take care of for you.

7. **Bridge Pattern**



The [**bridge design pattern**](http://www.devlake.com/) allows you to separate the abstraction from the implementation. In the [**bridge pattern**](http://www.devlake.com/), there are 2 parts - the first part is the Abstraction, and the second part is the Implementation. The [**bridge pattern**](http://www.devlake.com/) allows the Abstraction and the Implementation to be developed independently, and the client code can access only the Abstraction part without being concerned about the Implementation part.

**Where We Can Use It ?**

It’s hard to imagine event-driven programming without bridges. But new JavaScript programmers are often caught up in the functional style of event-driven development and they forget to write interfaces—even for difficult operations. It’s usually simple to diagnose where a bridge should be included. Say, for instance, that your code looks like the following:

$('example').onclick = function() {

new RichTextEditor();

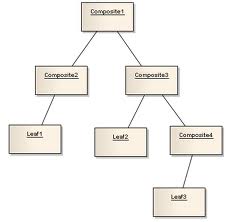
};

Nowhere does this tell you where the editor is going to show up, what the configuration

options are, or how to modify it. The key here is to make your interfaces “bridgeable” and in

fact, adaptable .In real life, this is critical to the construction of cities and the integration of roads within them. Neighborhoods are equivalent to modules, and roads are like the methods that connect them. The usability of a road often affects the population of that region. Likewise, the interface you offer clients will most likely affect its popularity.

8. **Composite Pattern**



The [**composite design pattern**](http://www.devlake.com/) allows you to set up a tree structure and ask each element in the tree structure to perform a task. A typical tree structure would be a company organization chart, where the CEO is at the top and other employees at the bottom. After the tree structure is established, you can then ask each element, or employee, to perform a common operation.

**Where We Can We Use It ?**

Objects within a composite are very loosely coupled. As long as all objects within a composite

implement the same interface, moving them around or interchanging them is a trivial

operation. This improves code reuse and allows easier refactoring. Composite objects make excellent hierarchical structures. Every time you execute an operation on a top-level composite, you are essentially performing a depth-first search on the entire structure to find the nodes. All of this is transparent to the programmer instantiating the object. It is very easy to add, remove, and find nodes within the hierarchy.

**Implementing Factory Method pattern and Observer Pattern in Javascript**

**EXPLANATION-:**

I am using Factory pattern and Observer Pattern and displaying how these patterns interact in the program.

Factory Pattern -:

The **factory method** pattern is a creational design pattern which does exactly as it sounds: it’s a class that acts as a factory of object instances.

Observer Pattern-:

In the Observer Pattern, a subject can have a list of observers that are interested in it’s lifecycle. Anytime the subject does something interesting, it sends a notification to its observers. If an observer is no longer interested in listening to the subject, the subject can remove it from its list.

In this program , the objects that I have created are :

1. Creator

2. AbstractProduct

3. Concrete Product

In this JavaScript example the Factory object creates four different types of employees. Each employee type has a different hourly rate. The createEmployeemethod is the actual Factory Method. The client instructs the factory what type of employee to create by passing a type argument into the Factory Method.