Design Pattern for Web services

Sanjay Krishna Anblagan
University of Massachusetts Lowell

Abstract

With the advent of Object oriented concept, duplicating the real world scenarios, insistence for application re-usability and efficiency has increased particularly in web-domains. So rather than knowing the technology, the programmer needs to follow method of implementation which assures long term usage, editable, re-usable, easy maintenance. This paper puts forth widely used Design patterns in web-service and the key OOP concepts in their implementation.

Keywords decorator, observer, composition, wrapper, interface
Introduction [4]

During past two decades growth and transformation of information on the web has been enormous. This has put lot of responsibility on developers to consider reusability, flexibility, management problems, editable and easy updatable. This paved the way for the rise of Design patterns, which gives architecture for common problems like state changes, adding new features to page, and change of algorithm on the fly. So before getting into design patterns it’s necessary for the developers to have knowledge on Object Oriented programming and its practice. Considering this fact, this paper initially explains some key concepts widely used in design patterns of web development followed by two widely accepted standard design patterns for web designing; Decorator pattern for client side and Observer pattern for server side discussed with class diagram. Finally as special note in visualization field of web technology, certain highly recommended design patterns which are derived from basic design patterns are explained with proposed structure diagrams.

1. Key OOP concepts used in these web designing patterns [1,2]

1.1 Wrapper

The downside of inheritance is that the subclasses inherit all the superclass features. Certain times we don’t require all the “stuff” from a superclass. If we want to add functionality to an object, subclassing may be unnecessarily burdened by unused functionality. This is where wrapping comes handy. Wrapping allows one to use another characteristic without extending either the wrapped object or the object doing the wrapping. Consider this example from AS3.

```javascript
var k:uint = 5;
var s:string = “7”;
var w:uint;
w = Number(s);
trace (k+w);
```

Here in this, the Number acts as a wrapper, instead of inheriting entire properties of the string it takes the numerical property of string using wrapper Number and uses it in execution. So these classes are termed as wrapper class.

1.2 Program to Interface over implementations

To understand why programming to the interface is encouraged, we need to understand the idea of flexibility and the issue of managing dependency in large problems. The need for flexibility leads to the need to manage dependency. When the code depends on a specific implementation and the implementation changes, our client dependency leads to undesirable results. The solution for this situation is to separate the design from implementation, in that process we separate client form the implementation as well. Thus interface decouples implementation from design.
1.3 Favor object composition over class inheritance

Let’s go with an example to understand this concept. The line `private var dog:Dog;` keeps the `reference` to `Dog class` in its class definition. That line is the basis of composition. The composition class now `Has a - base Classes` Dog, Bird, Fish. In this particular implementation, the composition class creates an `instance` of `base classes` in its `constructor`. Finally, a public function, `execute( )`, `delegates` the work back to `base classes`.

```
Package {
  public class Composition {
    private var dog:Dog;
    private var bird:Bird;
    private var fish:Fish;
    public function Composition() {
      dog=new Dog();
      bird=new Bird();
      fish=new Fish();
    }
    public function execute() {
      dog.BARK();
      bird.FLY();
      fish.SWIM();
    }
  }
}

Package{
  public class NewSpecies extends Sprite {
    private var composition:Composition;
    public function NewSpecies() {
      composition=new Composition();
      composition.Execute();
    }
  }
}
```

Suppose I want a species which needs to have the capability of swim, fly and bark alone instead of all the capability of fish, bird and dog. Composition comes handy by providing reference to bird, fish and dog class. Execute only the dog barking, fish swimming and bird flying capability whichever class refers them. So here NewSpecies class reference composition. Thus by favoring composition over inheritance unnecessary methods can be avoided.
2 Design Patterns

2.1 Decorator Pattern [1,2]

Consider a scenario – A class is created and released to the world. With the idea,
- Whoever uses it they can easily build upon it and release to outside world?
- Or they can use the extension that makes sense to.

Let’s go with an example, a class beautifies a text. A Yahoo developer might want to add a feature (a decorator) that also adds an exclamation after the word Yahoo (Hola!). A Spanish-speaking developer might add a feature where the exclamation sentences have the flipped exclamation mark before them. ¡Hola! Some might add other language-specific functionality. At the end, a new user of the class should be able to easily use the available features he likes, and leave out rest. (Hola…., Hola!!!!!!). So this is the situation where the decorator pattern comes handy. It gives the developer a model where he can reuse the base elements of the model and add, remove or replace components.

![Diagram of Decorator Design Pattern](image)

**Figure 1 - Decorator design Pattern**

The Decorator outlined in the class diagram notation shows: a component and a decorator. The component represents the base that is to be decorated, and the decorator is the abstract class for the concrete decorations. The concrete component (Hola) is what’s actually decorated, and the concrete decorations (..., !!!,) are the actual decorations. All the concrete decorations and the concrete component are subclassed from the same source (component class).

**Key OOP Concept**

**Wrapper**

While using the Decorator class, components are wrapped in decorators. The wrapping class is a concrete instance of a decorator, and the wrapped class is an instance of the concrete component class. Thus, the concrete component now “contains” the characteristics of the wrapper, but the characteristics are not inherited.
2.2 Observer Pattern [1,2]

In applications where a single source of information needs to be broadcasted to several different receptors, using a single source in the design makes more sense than having several different sources getting the same information by repeated calls to the data source. For example, in using a web service that sends out stock quotes, setting up your application to receive the information in a single source, and then sending out that information from that source in your application, is more efficient than having each instance calling the information separately from the web service.

![Figure 2 - Observer Pattern](image)

The Observer outlined in the class diagram notation shows two key elements: a subject and an observer. The subject interface provides the method for notifying the observers of state changes. We can break down the diagram into its four parts—subject, observer, concrete subject and concrete observer. The Subject interface sets up connections between itself and the observer, by establishing methods (subscribe and unsubscribe) that connect and disconnect the observer from the notification process. (ie.)Every single Subject method references the Observer. The Observer interface (update method) references the notification process from the subject. The ConcreteSubject provides a property (stateWork) for holding the state about which the observers are notified. Finally, through the update method, the ConcreteObserver keeps a state value consistent with the subject, and, through a state property, stores the current state.

Key OOP Concept

Composition

The Observer design pattern uses object composition instead of inheritance. Here the concrete subject will have the reference to all the classes subscribed for notification and execute only the required method in the observer class.
Derived Design Patterns

3.1 Reference Model [2,3]

This model achieves separation of data and visual models which is required for multiple visualizations of a single data source. In addition to this, to enable different views of a single visualization it separates visual models from displays.

The development of an information visualization application requires data management, visual mappings, use of computer graphics and different interactions. The re-usability, complexity and extensibility of these software architectures require the correct division of the above tasks. Therefore this reference model serves to be a template for those application that separate data models, visual models, views and interactive controls.

Its structure can be understood from the figure below. The datasource component loads the data to be visualized; this can be achieved using a formatted file reader or a database connectivity interface. One or more datasets can be registered using visualizations. As a result abstract data can be filtered from visual attributes such as size, shape, location allowing the creation of a single abstract dataset that can be used in several visualizations. Visual items can be created which represent a visually interactive object.

![Figure 3 – Reference Model](image)

The remaining three components, namely the View, Control and Visualization make use of the Model-View-Controller pattern as a result of which the visual model can be represented in more than one view, with the inputs by the user being processed by controls that have the capacity of affecting any level of the system.

**Design Pattern used**

*Factory Method pattern* → DataSource components to instantiate objects without specifying concrete classes.

*Observer pattern* → to pass update notifications between objects.
3.2 Data column Model [2,3]

Data tables or relational models are the most commonly used data representation in visualizations, with every row representing a single data record and each column, individual features of the same record. Database management systems store such data in a row-major order; however information visualization systems manage datasets in the main memory and therefore require a design pattern for internally segregating the data by row or column.

A column based approach has the advantage of simplifying data management wherein columns can easily be added and removed, shared across tables, each column implementing its own data management strategy for more efficient store of data. The data column comes to be extremely useful when one desires representational flexibility and extensible schemas.

Columns when aggregate together can be used to represent tables. Columns can also be shared across different tables with updated notifications provided. In addition, table schemas can be changed with minimum overhead by simply adding or removing column instances from the table. Handling of the column access can be done in several ways. Both a column and row manager can be maintained which maintains a map between the row numbers and column numbers.

Design Patterns used

*Observer pattern* → to propagate updates and facts.
*Factory Method Pattern* → to instantiate polymorphic data columns.
3.3 Cascaded Table Model [2,3]

Many times there arise situations when it is essential to extend the contents of a table with additions. To exemplify, creation of visual abstractions which add additional visual attributes such as color to an existing data model. This table helps in providing a solution to the problem of re-using non-changing properties among different displays in which only one property is being changed. The data table is only extended and not modified.

![Figure 5 – Cascaded Table Model](image)

It subclasses Table and also maintains a parent table. If a certain requested column is not found in the child table, the request is sent to the parent table. Updates are relayed from the parents to the child tables, thus achieving coordination and Cascaded Tables might contain data columns that shadow columns in the parent table.

There exists a requirement of a row manager that can coordinate between the child and parent tables. For example, when the cascaded tables need to provide filtered views of the parent, such management is essential. Cascaded tables help in providing visual abstractions enhancing a dataset with visual properties. As a result such a table helps in providing an access into the visual properties and the underlying data through a unified interface. Cascaded tables also provide a means to re-use visual properties within a display and between displays.

**Design Pattern Used**

*Decorator pattern* → extending a parent table with additional data.
3.4 Scheduler [2,5]

Animation, for example is a dynamic visualization that makes use of time-sensitive recurring operations. It requires constant redrawing of the display at regular intervals and constantly updating visual properties. The Scheduler helps in providing a solution to this problem and supports extensibility.

One consideration in the Scheduler, is its handling of concurrency. A commonly used approach is its dispatch loop to run in a dedicated thread. Another approach entails its running within the user interface event loop; this frees programmers from explicitly tackling this issue. However in the case of greedy activities with long running “run” methods, the other components of the visualization become unresponsive.

As described in the figure, custom operations are made by subclassing the abstract Activity class and using the “run” method. It has a specified start time, duration and a step time which decided the time interval between repeated invocations.

Design Pattern used

*Template pattern* → Operations are invoked in the same order at regular intervals of time
3.5 Operator Model [2,5]

This design solution can be used for achieving a certain amount of granularity in for the various components in the visualization system. This is necessary while designing object-oriented visualizations with the result being a library of interactive elements such as scatterplots, treemaps etc which can be added any time to the user interface. This could be interpreted as including different visualizations into monolithic widgets. Examples of operators include layout algorithms, distortion mechanisms and visual encodings. New visualizations can be made using pre-existing operators allowing re-use of visualization techniques, or by constructing using new ones. Customization tasks may also be offered which allow the clients to modify the set of operators that are being used for the visualization.

As shown in the figure the operator has a single decided method that performs the operation making it simple to create new, general purpose operators. They might maintain a reference to the visualization or may use several parameters to operate. The operator will have different parameters depending upon its function. Operators also make use of other design patterns such as Expressions and Scheduler allowing recurring invocations. Operators fit within the data flow model and data state model. However the author has focused on the data flow model because of its prevalence in information visualization.

**Design Patterns used**

*Strategy pattern* → to decide on the set of actions to be performed.

*Flyweight pattern* → to reuse existing operators.
Conclusion

With the advent of light weight toolkit scripts like Jquery and mojotool, and ActionScript3 of Adobe replicating almost most of the technology of system software, we can expect lot more design pattern in coming days. Though lot more design pattern are in practice I went with basic design pattern used by web-developers more frequent and took the field of visualization of data in web-technology to explain how basic patterns stands as basic structure in derived patterns.

Reference

Books

Publications & Web
[5] Design Patterns for Rapid Visualization Prototyping, Mark Giereth and Thomas Ertl Visualization and Interactive Systems Institute, University of Stuttgart.