2015
Pages 1–4

Structural Design Patterns

Author: Ahmed Khattab

Seminar: Javascript Patterns and Anti-patterns [SS15]

Supervised By: Guy Yachdav

ABSTRACT

This paper provides an overview of structural design patterns in software engineering, with special focus on Javascript based applications. A general introduction of structural patterns followed by a detailed elaboration on three of these patterns is presented. Additionally, the paper includes refactoring use cases of actual projects - publicly available on GitHub - to illustrate the use of structural patterns.

1 INTRODUCTION

In their design patterns book [1], Gamma et al. classified software design patterns into three major categories based on their *purpose* (the purpose of a design pattern basically defines what is used for). These three categories namely are: *Behavioral* patterns, *Creational* patterns and *Structural* patterns.

2 STRUCTURAL PATTERNS

Structural patterns refer to a category of design patterns which are concerned with how classes and objects are composed to form larger and more complex structures. Structural patterns hence provide simple and organized ways for realizing relationships between objects. [1] [2]

A simple example showing how structural patterns can help in realizing relationships between objects is the *Adapter* pattern. An adapter is basically used to make a component A (possibly a legacy subsystem) conform with another component B, this is achieved through wrapping the first component A with an interface to which B can easily connect and communicate.

Another example is the composite pattern. The composite pattern can be applied in situations where two types of objects can be distinguished: primitive objects and composite objects, with both types exhibiting similar functionality. Composite objects are composed of a number of primitive components, or may be recursively composed of other composite components (e.g. in a file system, a directory can contain files or other directories).
In this case the composite pattern helps in defining a uniform way for modeling both composite and primitive objects.

Other examples of structural patterns include: proxy, façade, decorator, bridge and flyweight design patterns. In the next sections, three of these patterns are chosen and presented thoroughly (in the context of Javascript applications) namely the decorator, the façade and the proxy pattern, along with refactoring example showing these patterns in action.

3 THE DECORATOR PATTERN

3.1 Motivation

The main purpose of the decorator pattern is to allow the addition/modification of functionality of an object in a dynamic way. These functional additions or modifications can be referred to as *decorations*.
By allowing the application of such *decorations* to a specific object and not to the class itself, the decorator pattern provides a flexible alternative to the normal sub-classing and inheritance approach.
Another important feature of the decorator pattern is that it allows the combination of more than one decoration, giving the freedom to the developer to choose how the combination is actually done.

In Javascript, the addition of new functionality to an object is not a problem since objects are mutable. However, the decorator pattern provides a more organized way for defining and reusing such additions.

3.2 Common Scenarios

One of the common situations for using the decorator pattern, is when there is a special functionality modification that needs to be done for an object, without modifying other parts of code relying on the base - unmodified - object.

Another common situation, is when the application depends on a big set of dynamic options that makes sub-classing impractical and a pain to manage. A possible example of such situation is in the case of a game where there are several roles the player can choose from, each with different capabilities; modelling such a situation using sub-classing would result in a big inheritance tree that is hard to manage and inflexible to change.

3.3 In Details

As shown in the UML diagram in Figure 1, the interface *Decorator* provides an abstract definition for all decorators. Each concrete decorator should provide its own flavor for implementing the `operation()` method or for adding new operations. Concrete decorators can be designed to be stacked or combined together as mentioned previously.

In the context of Javascript, the decorator pattern can be realized generally in two ways. The first method uses inheritance; where

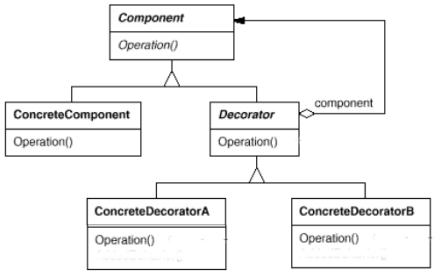

Fig. 1. Decorator: UML [1]

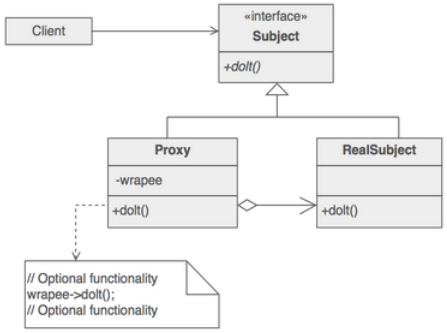

Fig. 2. Proxy: UML [3]

each time a decoration is applied, a new object is created to wrap the decorated object while adding the new/modified functionality (basically the same idea depicted in Figure 1).

The second way of realizing the decorator pattern makes use of the dynamic features of Javascript and avoids the use of inheritance.[4] In this case, a list of different decorators is created, this list is a global list that can be associated with the class itself. When the object is decorated with one of the options in the aforementioned list, the chosen decorator is pushed into another list that is specific to each individual object.
Later on when the decorated behavior is called, the decorations list of the object is checked to apply any decorations that have been pushed into it.

It is worth mentioning also that the idea of decorating objects with extra functionality can be realized through jQuery through the function `jQuery.extend()` which allows the dynamic extension of an object.

4 THE PROXY PATTERN

4.1 Motivation

The proxy pattern is based on the idea of having a proxy that sits between the clients of an object and the object itself. The proxy constitutes a level of indirection that controls the access to the target object, delegating requests and wrapping responses, while protecting the target object from unnecessary inefficient access.

4.2 Common Scenarios

The proxy pattern is typically useful to handle the access to an object that is expensive to initialize. In this case, the proxy is used to postpone the execution of costly operations until they are necessary.

Another scenario is when the proxy is used to provide a transparent access to a remote object, making it appear to be local. The proxy can as well provide caching mechanisms for reducing the number of remote calls.

4.3 In Details

As shown in the UML diagram in Figure 2, both the *Proxy* and the *RealSubject* implement the same interface. The client requests are however passed first to the *Proxy* instance, which in turn decides whether it is necessary to delegate the request to the *RealSubject* or not.

5 THE FAÇADE PATTERN

5.1 Motivation

The main purpose of the façade pattern is to provide a unified and simple interface for one or more complex subsystems. The façade pattern hence helps in making those subsystems easier to understand and to use. Also such a simplified interface helps in reducing the number of inter-dependencies between these subsystems and their clients, and hence achieving more decoupling.

5.2 Common Scenarios

The façade pattern can be useful when a layered structure of the system is wanted, where each layer is hidden behind a façade that provides the entry point to the components of this layer. [1]

Another scenario is when the application involves interactions with a set of poorly designed APIs. A façade can hence be used to wrap these APIs, providing a better interface for the rest of the application to interact with.

5.3 In Details

As shown in Figure 3, the façade interface is added to handle the client requests to the body of subsystems encapsulated by it. The façade handles all explicit interactions to the internal classes of the different subsystems, while the outside clients do not need to know about the details of these interactions.

6 REFACTORING EXAMPLES

In order to show how different structural patterns can be used to tackle design problems, two GitHub projects are used as a use case for analysis of possible problems and refactoring using the structural patterns discussed in the previous three sections.

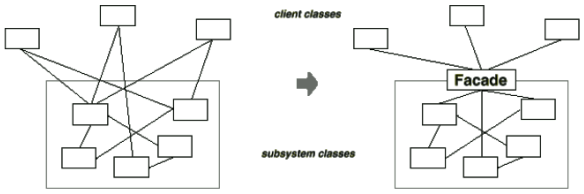

Fig. 3. Facade: illustration [1]

6.1 The Weather Searcher

The weather searcher[1] is a simple Javascript application that provides access to location based weather information through communication with a weather web service.
The user of the application can get weather information based on his current location or based on another location that he specifies through a text field.

If the user enters the name of a location that is ambiguous (i.e. if there are several possible matches for the entered location), the web service json response will include a list of possible matches for the location the user entered, which are then displayed for the user to choose from.

The response of the web service can hence be either the actual weather information for a certain location, a list of possible location matches or otherwise an error.

*6.1.1 **Analysis**:*

The original implementation of the application relies on a single javascript function that acts as callback for all the different ajax requests that are done in the different scenarios.

The callback function is responsible for receiving the response from the web service, as well as parsing and rendering it. Given that the response can be either actual weather information or a list of possible location matches or an error, different checks are done in this function in order to handle each case and render the view accordingly.

*6.1.2 **Drawbacks**:*

Possible drawbacks of the original code are:

- The tight coupling between the view, the control and the model, since one function is responsible for receiving and handling the different responses of the web service and for rendering the view.
- The web service is called every time the user enters an ambiguous location and hence the same list of possible matches is fetched every time from the web service, although the information is static and caching it would save unnecessary calls to the web service.

[1] https://github.com/flamingveggies/weathersearcher

*6.1.3 **Refactoring**:*

To tackle the aforementioned drawbacks, refactoring of the code is done on two steps using the *proxy* and the *decorator* patterns.

A class `API` is introduced to play the role of a *proxy* between the application and the web service. This class provides different methods corresponding to the different calls that are made to the web service. Each method can take as an argument a callback that is called after the ajax request is done and the response is returned.

Additionally this `API` class is responsible for managing a cache, modelled as a simple array, that stores the response whenever it contains a list of location matches. This cache is checked for cache hits before a new request to the web service is made.

A class `BasicResponse` is also added to wrap every json response that is returned from the web service. The class `BasicResponse` also provides the method `render()` responsible for visualizing the response.

As mentioned previously, the web service response can take several forms (e.g. actual weather info or list of location matches) and hence each form is rendered differently, hence comes the use of the *decorator* patter.

In a decorator pattern sense, the different renderings are analogous to different decorators that are dynamically chosen according to the nature of the returned response.

The different render functions are added to a list of decorators (e.g. `locationResonse` is the decorator that applies the rendering for a list of locations), as shown in this code snippet:

```
decorators = {};
decorators.locationsResponse = {
  render: function(data){
  //do custom rendering for locations
      suggestions list...
  }};
```

Whenever the returned response is actually a list of locations, the function `decorate()` is called on the `BasicResponse` object with the parameter "`locationResponse`", what happens next is that the attribute decorator of the object is set and during rendering this decorator will be used instead of the default rendering function.

```
var weatherResponse = new BasicResponse(data);
if(data.locations != '') {
  weatherResponse.decorate('locationsResponse')
      ;
}
weatherResponse.render();
```

It is worth mentioning that the `render()` is always called in the same way but what happens inside depends on whether a decorator is set for the object or not, this illustrates how the object functionality is changed dynamically at runtime.

6.2 Pacman

The second refactoring example used to demonstrate the use of structural patterns is an HTML5/Javascript implementation of the game *Pacman*[2].

The refactoring done focuses on a specific class of the game which is the class `Ghost`. This class is responsible for all functionality and rendering of the enemy characters in the game.

Visually, the ghosts normally have different colors and have a special state (the weak state) that they enter whenever pacman eats one of the big beans at the corners of the game board (rightmost shape in Figure 4). More small details are also taken into account like changing where the ghosts' eye balls are looking according to the the direction of movement of the ghost (i.e. looking left when moving to the left, etc..)

Fig. 4. Pacman ghosts: illustration

6.2.1 *Analysis:*

In the implementation, the `Ghost` class included a huge `draw()` method which handles everything related to rendering the ghosts in their different states. In a very repetitive style, the function handles drawing every element of the ghost shape (eyes, mouth, legs, etc..), for example for the eyes the same lines of code are repeated 4 times to handle drawing the eyes for the 4 different directions of movement.

6.2.2 *Drawbacks:*

The `draw()` function is hard to maintain and to understand. Many lines of code are unnecessarily repeated, where the same details are done with only a small change in parameters. There is a potential for reducing the complexity of this function through dividing it into smaller reusable parts.

6.2.3 *Refactoring:*

The refactoring is done by introducing several façades which encapsulate the details of drawing different elements of the ghost shape, and at the same they are reusable and flexible to changes in parameters. The `draw()` function is restructured to focus more on handling the logic behind the rendering, leaving the details of the rendering to the added façades

As an example, a function `drawEyes(xOffset, yOffset)` is added to handle drawing the eye balls. By passing different offset parameters, the function makes it easy to draw the eye balls in their correct position according to the movement direction. Similarly, other functions are added like `drawMouth()` and `drawLegs()`.

[2] https://github.com/bxia/Javascript-Pacman

7 DISCUSSION

The decorator pattern helps in *decorating* objects with one or more additional or alternative functionality. However, excessive use of decorators and specially without thorough documentation is not advised, since the use of decorators introduces a lot of small objects, which makes their initialization and maintaining their conformance to a common interface a hard task. Moreover, things can get more complicated if there is a big set of inter-related decorators.[5] [2]

The proxy pattern provides a level of indirection that helps in regulating and optimizing access to expensive objects. However, in the case of proxies for remote objects, one must be careful that while making access to a remote object seem transparent, a developer who does not know about the underlying remote access costs can run into some inefficiencies as shown in the following code snippet:

```
1 if (account.getBalance() > 0 && account.
      getBalance() < MAX) {
2   transferAmount(account.getBalance() / 2);
3 }
```

In this example, the `getBalance()` function encapsulates possibly a remote call to fetch the account balance data from a database, not knowing this can lead to redundantly calling the function as shown in the example, while it would be more efficient to call it once and store the value in variable before the if-condition.

The façade pattern provides an abstraction to encapsulate more complex interactions to other parts of code or other subsystems, enhancing the structure, the readability and the maintainability of code. However, one must be aware about the trade-off between adding more abstraction layers and the increasing performance costs of adding these abstractions.

8 CONCLUSION

Structural design patterns offer a set of solutions for realizing relationships between entities in the application, while keeping the code well-structured, understandable and flexible to change.

Structural patterns are in some cases intuitive as in the case of the façade pattern, but can also be more involved in terms of how they can be realized as in the case of the decorator pattern.

It is worth mentioning also that the application of these patterns usually include a trade-off of pros and cons, which hence need to be analyzed and studied according to each individual situation.

REFERENCES

[1]E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 1994.

[2]A. Osmani. *Learning JavaScript Design Patterns - a JavaScript and jQuery Developer's Guide.* O'Reilly, 2012.

[3]Sourcemaking. Design patterns explained simply. `https://sourcemaking.com/design-patterns-book`, 2015. [Online; accessed 8-May-2015].

[4]S. Stefanov. *JavaScript Patterns*. O'Reilly Media, 2010.

[5]M. Zaikin. A selected list of benefits and drawbacks of patterns from the gof book. `http://java.boot.by/scea5-guide/ch07s03.html`, 2007. [Online; accessed 8-May-2015].