Structural Patterns

Chrysa Papadaki, Nishant Gupta
Department of Computer Science, Boltzmannstr. 3, 85748 Garching

Associate Editor: Guy Yachdav

ABSTRACT

Structural design patterns provide a way of solving a general design problem by identifying the relationships between involved entities. This paper presents the introduction to structural design patterns and describes the way these patterns can be applied in JavaScript applications. To illustrate the use of structural patterns, two JavaScript projects that already exist on GitHub [5] are critically analyzed. In each project source code has been optimized by applying one or multiple structural patterns.

1 INTRODUCTION

In their design patterns book [2], Gamma et al. classified software design patterns into three major categories based on their purpose. These three categories are: Behavioral patterns, Creational patterns and Structural patterns. The focus of this report is to elaborate on the category of structural design patterns.

A structural design pattern serves as a blueprint for how different classes and objects are combined to form larger structures. The composition between the modules or components which allow one to develop or build larger systems. Each one has a different purpose and they all involve connections between objects.

In addition, in some sense, structural patterns resemble the simpler concept of data structures. However, they also specify the methods that connect objects, not merely the references between them. A structural design pattern also describes how data moves through the pattern. not only how data is arranged in the structure.

In general these patterns help us to structure the objects, the relationships and inheritance between classes. They help in sharing data between similar objects by using Flyweight [6], which allows efficient sharing of objects to save space, when there are many instances, but only a few different types. They allow us to treat single and composite objects uniformly and thus reduce the complexity of code which leads to a less error prone implementation, using Composite pattern [6]. We can change the behavior of an object at runtime, by using Decorator pattern [6] or hide the complex code or interface behind a more simplified one by using Facade pattern [6].

2 STRUCTURAL PATTERNS

There are multiple structural patterns that can be applied in any JavaScript project. In this paper the following structural patterns have been reviewed [2]:

- *Proxy pattern*
- *Decorator pattern*
- *Facade pattern*
- *Composite pattern*

2.1 Proxy Pattern

The Proxy Pattern focuses on providing a surrogate or placeholder object, which references an underlying subject. It provides the same public interface as the underlying subject class and introduces a level of indirection by accepting requests from a client object and passing these to the real subject whenever it is necessary. In practice, it adds a wrapper and delegation to protect the real component from undue complexity.

2.1.1 Applicability: There are four common situations in which the Proxy Pattern is applicable.

- A *Virtual proxy* provides a simplified version of a complex object. Only when the detail of the object is required is the main object actually populated, providing a form of lazy initialization.
- A *Remote proxy* provides a local object that references a subject object in another location, generally over a network connection.
- A *Protective proxy* adds a layer of security to the underlying subject object. It could add methods or properties that allow the client object to provide appropriate authentication before allowing the data to be returned.
- A *Smart proxy* interposes additional actions when an object is accessed. It adds extra functionality to the calls to the real object's members. This functionality is often invisible to the client object.

2.1.2 Structure: The structure of Proxy Pattern is illustrated by the figure 1. By defining a Subject interface, the presence of the Proxy object standing in place of the RealSubject is transparent to the client. This demands the same public interface in both Proxy and RealSubject classes i.e. +doIt() operation. The Proxy object references to RealSubject and delegates the request to it only in case it is needed.

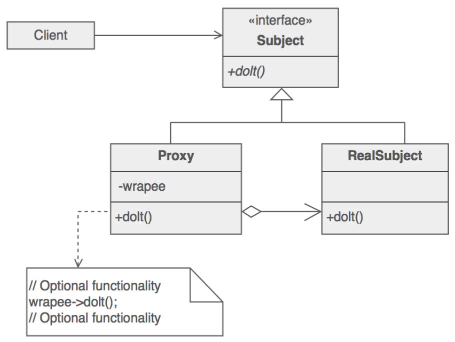


Fig. 1: Generic structure of the Proxy pattern [6]

2.1.3 Benefits: A Cache Proxy improves the performance of the underlying object's members when they perform long-running tasks that return seldom-changing results. The Remote proxy also can ensure security by installing the local code proxy (stub) in the client machine and then accessing the server with help of the remote code.

2.2 Decorator Pattern

The Decorator pattern extends an individual object's behavior dynamically. It adds or removes functionality to/from an object at runtime. In practice, It provides a more flexible way to add responsibilities to a class than by using inheritance, since it can add these responsibilities to selected instances of the class. Multiple decorators can add or override functionality to the original object.

2.2.1 Applicability: It is applied for adding behavior or state to individual objects at runtime. This could be done by using subclassing, however inheritance is not feasible because it is static and applies to an entire class. In addition, it can remove these (encapsulated) responsibilities again without affecting other objects. For instance, this pattern could be used for Extending capabilities of a Graphical Window at runtime.

2.2.2 Structure: Figure 2 depicts the structure of the Decorator Structural Pattern.The participants classes in the decorator pattern are the Component Interface for objects that can have responsibilities added to them dynamically. ConcreteComponent which defines an object to which additional responsibilities can be added, the Decorator which maintains a reference to a Component object and defines an interface that conforms to Component's interface and the Concrete Decorators which extend the functionality of the component by adding state or adding behavior.

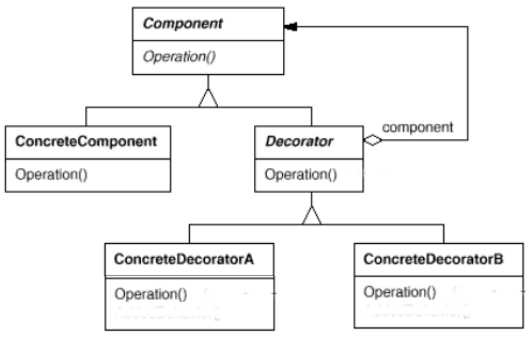


Fig. 2: Generic structure of the Decorator pattern [2]

2.2.3 Benefits: Given the above-mentioned, the major advantage hidden behind the Decorator pattern usage is the case where developers are confronted with a large number of independent ways to extend functionality. Especially when it is not easy to predict at design time which combinations of extensions will be used during runtime.

2.3 Facade Pattern

Facade design patterns provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use [2]. Facade pattern hides the complexities of the system by providing an interface to the client using which the client can access the system.

2.3.1 Applicability: Facade pattern should be used when it is required to provide a simple interface to a large and complex system. It can also be used when there are many dependencies between clients and the implementation classes of an abstraction. If multiple subsystems are to be layered, Facade classes can define entry points to each subsystem layer.

2.3.2 Structure: Typical structure of Facade usage can be depicted as shown in the figure 3. Here, subsystem classes implement respective subsystem's functionality and facade has the responsibility to delegate client requests to appropriate subsystem objects. Subsystem classes don't have any knowledge of the facade, but handle the work assigned by the Facade object.

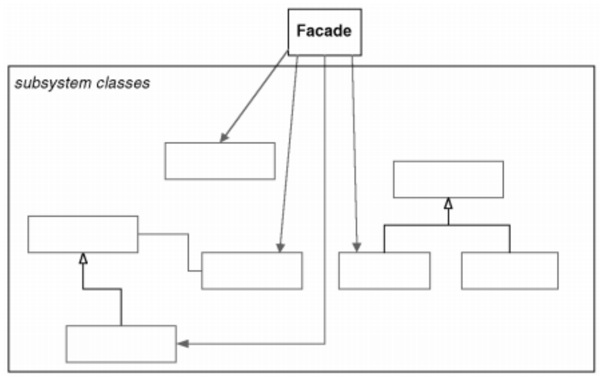


Fig. 3: Generic structure of the Facade pattern [2]

2.3.3 Benefits: By providing abstraction to clients from subsystem components, facade makes the subsystem easier to use for clients. Facade also promotes the concept of changing the components of a subsystem without affecting its clients, hence providing a weak coupling between the subsystem and its clients. Besides all these features, facade doesn't prevent applications from using subsystem classes if they need to.

2.4 Composite Pattern

Composite design pattern states that a group of objects should be treated in the same way as a single instance of object is treated. This implies that same behavior should be applied to an individual object or a composition of objects.

2.4.1 Applicability: Composite pattern should be used when part-whole hierarchies of objects have to be represented. Client should be able to overlook the difference between individual objects and composition of objects, hence treating all objects in the composite structure uniformly.

2.4.2 Structure: A typical Composite pattern structure looks as depicted in figure 4. Here, Component acts as an interface for objects in composition and defines the default functionality of classes. Leaf defines the behavior of leaf objects while Composite represents components having children. Client in the end has to access and manipulate objects in the composition through Component interface.

2.4.3 Benefits: Composite pattern provides an efficient way of defining class hierarchies consisting of primitive objects and composite objects. Since client can treat composite objects and individual objects uniformly, it makes client implementation much more simple. Adding new components becomes simple using composite pattern which proves overly general design of implementing composite pattern in a code.

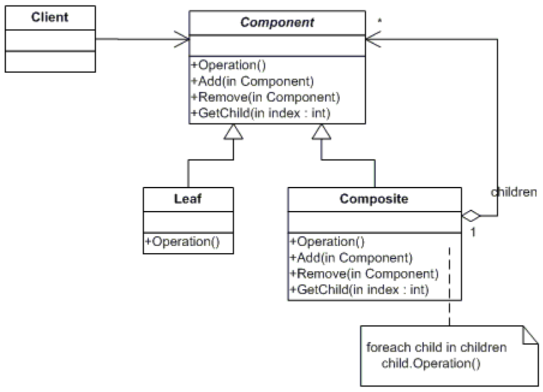


Fig. 4: Generic structure of the composite pattern [2]

3 REFACTORING EXAMPLES

This section provides overview of the two JavaScript projects that were found on GitHub [5]. Brief analysis of each project along with its limitations based on poor code structure is discussed. This is followed by "Refactoring" section where modifications done on these projects based on the use of structural patterns is shown.

3.1 songFinder

songFinder [3] is a typical song search engine that consumes Spotify [1] services. It offers a simple user interface where the user enters a song keyword and a list of songs is rendered. Each list item includes a song image, description and a redirect to media player button where user can listen to the song. Figure 5 shows this functionality.

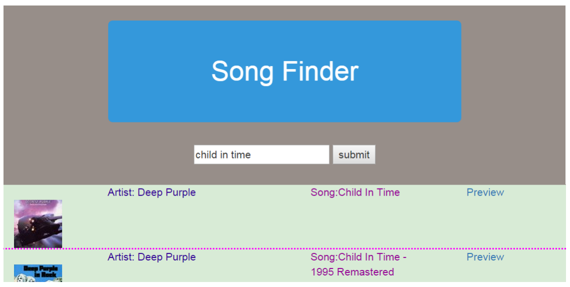


Fig. 5: songFinder - Song Search sample

3.1.1 Analysis: The original JavaScript implementation of songFinder (scripts.js) consists of the following main elements:

- *The Song object:* which is the data model,
- *The Jquery .on method:* which handles the form submit event. It is responsible for calling the spotify web service and triggering everytime a nested method invocation which results in the parsing and rendering of the model.

The methods display() and createSongsObj() are the one that participate in the nested method invocation and are responsble for rendering and parsing the data model respectively.

Taking the above into account, this code structure lacks readability and promotes difficulties in maintaining and extending the code. Handling of different server responses is not managed as well as caching.

3.1.2 Code Structure Limitations: The concrete drawbacks of the original implementation are:

- *No caching:* The application calls the web service every time the user presses the submit button for any input. Thus, every time the input leads to multiple song entries, a long list of songs is being fetched and rendered. This leads to performance issues.
- *Tight coupling:* There is tight coupling between the parsing and the rendering of the data model. Even though there is nested method invocation where the output of each inner method is input of the outer one, which makes code a bit more readable than all implementation being in one function, the handling of the different web service responses is not being managed. Therefore, a more clean and object orient way is needed to differentiate the result view based on the different server responses.

3.1.3 Refactoring:

Applying Proxy pattern: In order to introduce caching and improve the performance of the application, the SongWS and SongProxy classed were introduced.

- *SongWS* represents the real subject in the figure 6 and is responsible for calling the spotify web service and return the fetched data over the getSong() function.
- *SongProxy* represents the proxy entity in the figure 6 and is in charge of accepting the client object requests and forward to realSubject if the result of the specify song input is not in the cache. So, on the first call for a specific song, it caches the result and on every subsequent call it returns the result from the cache.

The figure 6 shows how the SongProxy which has the same interface with SongWS. i.e. getSong() handles the client object request and caches the result.

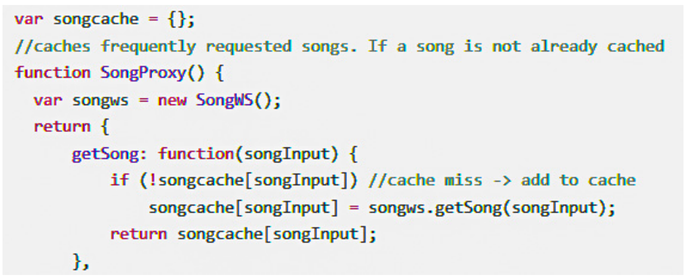

```
var songcache = {};
//caches frequently requested songs. If a song is not already cached
function SongProxy() {
  var songws = new SongWS();
  return {
      getSong: function(songInput) {
          if (!songcache[songInput]) //cache miss -> add to cache
              songcache[songInput] = songws.getSong(songInput);
          return songcache[songInput];
      },
```

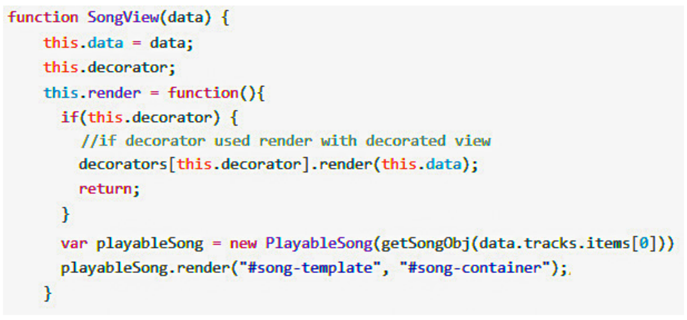
Fig. 6: SongProxy

Applying Decorator pattern: The need of handling different server responses dynamically led us to introduce a decorated object called PlayableSong and class SongView and the decorator songViews.

- *PlayableSong* extends the behavior of the base Song object by adding code in its render() function which adds the media controls and makes the song starts playing automatically. It was introduced to handle single song server result. Doing so, the user does not need to click on preview button.

- The decorator *songViews* is used for visualizing a list of Song objects.
- *SongView* class using the render function decides whether to use the render implementation of the decorator songViews or the base implementation of the PlayableSong for the single server result. It maintains the fetched data and it has the attribute decorator which can be defined via the public function decorate of the SongView class. This function is called by the display function when it decides that the data consists of a list of songs.

The Figure 7 shows how the SongView class can handle the fetched data by rendering in different views in a clean and efficient way.



```
function SongView(data) {
    this.data = data;
    this.decorator;
    this.render = function(){
      if(this.decorator) {
        //if decorator used render with decorated view
        decorators[this.decorator].render(this.data);
        return;
      }
      var playableSong = new PlayableSong(getSongObj(data.tracks.items[0]))
      playableSong.render("#song-template", "#song-container");
    }
```

Fig. 7: Decorator applied

3.2 drawr-bootstrap

drawr-bootstrap [4] is a nice and simple paint application which utilizes HTML5's canvas utility to draw on browser window. First screen of the application with a drawing sample is shown in figure 8

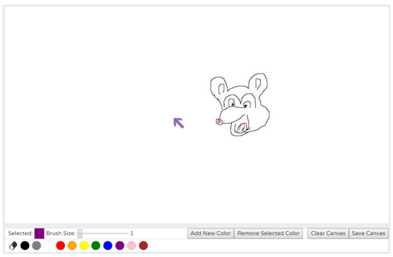


Fig. 8: drawr - Initial screen with drawing sample

Following are some of the features that this application offers to its users:

- *Customizable Drawing Tool*
- *Toolbar*
- *Brush/Eraser Thickness*
- *Color Palette*
- *Add new color*
- *Remove a color*
- *Clear canvas*
- *Save canvas*

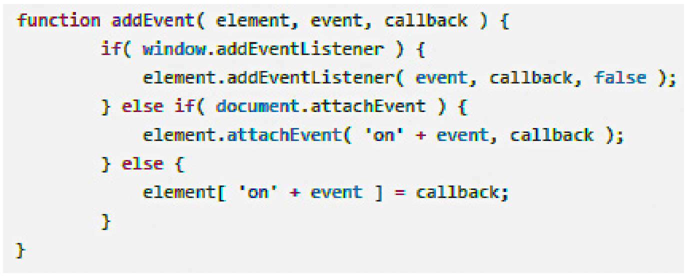
3.2.1 Analysis: The implementation of application consists of a large number of event listeners and corresponding event handlers. For example, 'click' event listener on each color to select that particular color for drawing. In the original version of application, all the event handlers are defined in a single function which is called when document is loaded. These event listeners and handlers are written in app.js which is the only JavaScript file present in whole source code.

3.2.2 Code Structure Limitations: Since all event handlers are defined in a single function of app.js, the code seems confusing to developers. Code abstraction, despite being one of the major code design goals of software engineering, has not been done in the original version. A block of code having a long, poor structure reduces the code clarity and readability. Besides this, browsers' compatibility has not been checked, that's why the application doesn't run on older versions of browsers like Internet Explorer 5.0 or Opera 1.0. Adding browsers' compatibility check to each event handler in the original block of code would enhance code complexity and will make it less presentable.

Another drawback in terms of code's structure is having a different approach for adding event handlers for an individual object and a composition of multiple objects. For a single element component, event listener id added by using JQuery's method corresponding to the event directly on the element selected. Whereas for composite component with a collection of list elements, JQuery's 'on' method is used with event name passed as first argument. In this way, single element components are handled differently than a composite element.

3.2.3 Refactoring: Original code of drawr [4] application has been refactored to remove limitations by introducing two structural design patterns- Facade and Composite.

Applying Facade pattern: A simple facade has been added that provides an interface to add event handlers to different elements, thereby providing a nice abstraction layer for clients. A function "addEvent" (see figure 9) is defined in the app.js file which is used every time a new event listener and handler has to be defined. Here "callback" is the event handler function that

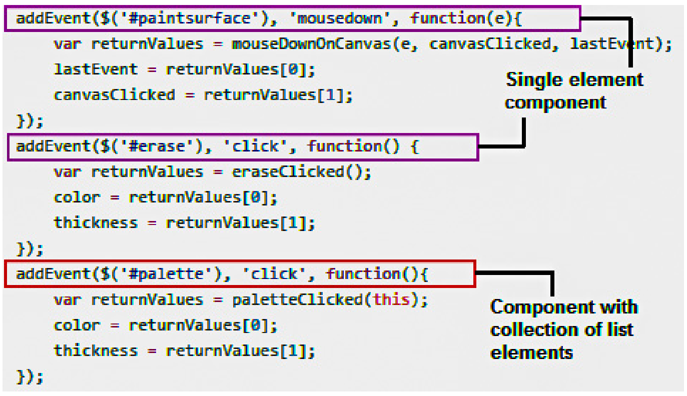


```
function addEvent( element, event, callback ) {
        if( window.addEventListener ) {
            element.addEventListener( event, callback, false );
        } else if( document.attachEvent ) {
            element.attachEvent( 'on' + event, callback );
        } else {
            element[ 'on' + event ] = callback;
        }
}
```

Fig. 9: A simple facade that masks the various browser specific methods

will be called when the "event" is triggered on the "element" component of HTML. All the event handlers are defined in a separate file 'eventHandler.js', which makes code clear and readable by providing abstraction to handler's implementation. Browsers' specific checks are added so that the application runs successfully on all browser versions.

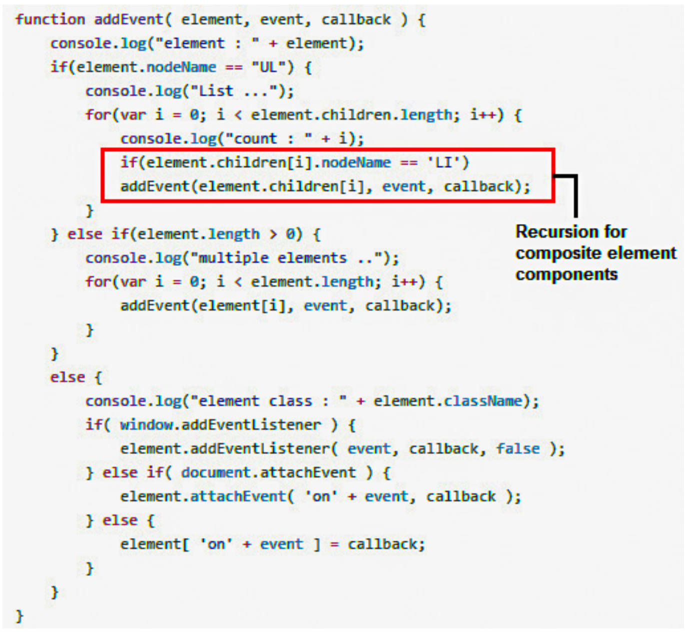
Applying Composite pattern: "addEvent" function defined for facade implementation has been extended to ensure that same kind of behavior is applied to individual element component as well as collection of elements. As depicted in figure 10, event listeners for individual elements (#paintsurface, #erase, etc.) are handled in same manner as done for component with collection of list elements (#palette).



```
addEvent($('#paintsurface'), 'mousedown', function(e){
     var returnValues = mouseDownOnCanvas(e, canvasClicked, lastEvent);
     lastEvent = returnValues[0];
     canvasClicked = returnValues[1];
});
addEvent($('#erase'), 'click', function() {
     var returnValues = eraseClicked();
     color = returnValues[0];
     thickness = returnValues[1];
});
addEvent($('#palette'), 'click', function(){
     var returnValues = paletteClicked(this);
     color = returnValues[0];
     thickness = returnValues[1];
});
```

Fig. 10: Same interface of addEvent provided to all kinds of element components for adding event listeners

The implementation of addEvent checks the kind of element passed. For a composite element, same method is called recursively for each leaf element component to provide uniform behavior in the end. Implementation for #palette element which contains a series of 'list' items is shown in the figure 11.



```
function addEvent( element, event, callback ) {
    console.log("element : " + element);
    if(element.nodeName == "UL") {
        console.log("List ...");
        for(var i = 0; i < element.children.length; i++) {
            console.log("count : " + i);
            if(element.children[i].nodeName == 'LI')
            addEvent(element.children[i], event, callback);
        }
    } else if(element.length > 0) {
        console.log("multiple elements ..");
        for(var i = 0; i < element.length; i++) {
            addEvent(element[i], event, callback);
        }
    }
    else {
        console.log("element class : " + element.className);
        if( window.addEventListener ) {
            element.addEventListener( event, callback, false );
        } else if( document.attachEvent ) {
            element.attachEvent( 'on' + event, callback );
        } else {
            element[ 'on' + event ] = callback;
        }
    }
}
```

Fig. 11: addEvent called recursively for each list item of composite component

4 DISCUSSION

Even though the applied structural design patterns improved the performance and the structure of the given implementation, there are remarks about the consequences which they might cause when they are overused. The Proxy pattern may imply a remote network call, which can potentially lead to an exception, or even return a different value each time, making the code inefficient. The Decorator pattern has a limitation that a decorator and its enclosed component are not identical. Thus, tests for object types will fail. The second drawback is that overuse of decorators can lead to a system with "lots of little objects" that all look similar to the programmer trying to maintain the code. Composite pattern although provides great code re-usability, but also has certain trade-off. Once tree structure is defined, the composite design makes the tree overly general. In specific cases, it is difficult to restrict the components of the tree to only particular types.

5 CONCLUSION

Structural design patterns provide a good way to design object-oriented software, whose code quality would depend on developer's experience with these patterns. They make a system less complex by providing the abstraction at a higher level. Describing a system in terms of the design patterns used in the implementation makes it much easier to understand, maintain and extend. But in the end, a system would be robust only if these patterns are applied carefully to make the code more valuable, rather than hiding its limitations. Thorough analysis of the situation should be done prior to fitting a design pattern to it.

REFERENCES

[1]Spotify Web API. https://developer.spotify.com/web-api.
[2]Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.
[3]Goodbedford. songfinder, https://github.com/goodbedford/songFinder, 2015.
[4]Merritt Lawrenson. flamingveggies/drawr, https://github.com/flamingveggies/drawr, 2015.
[5]GitHub Where software is built. https://github.com.
[6]SourceMaking. https://sourcemaking.com/design_patterns.