Structural Patterns

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

Associate Editor: Guy Yachdav

ABSTRACT

In order to demonstrate the application of general JavaScript patterns several projects on GitHub have been reviewed. In each project some code parts have been optimized by applying one or multiple structural patterns.

1 INTRODUCTION

————- Introduction to structural patterns ————

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

——- Proxy Pattern ——

2.2 Decorator Pattern

——- Decorator Pattern ——-

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 and 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 1. 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.

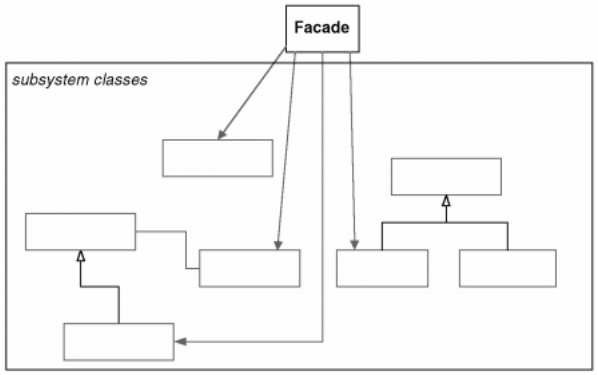

Fig. 1: Generic structure of the Facade pattern [2]

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 2. 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.

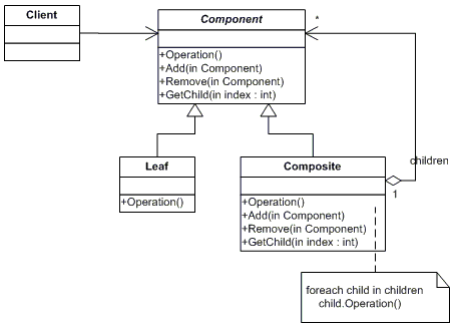

Fig. 2: Generic structure of the composite pattern [2]

3 REFACTORING EXAMPLES

3.1 songFinder

3.2 drawr-bootstrap

drawr-bootstrap [1] 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 3 Following are some

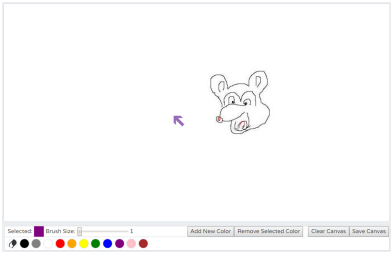

Fig. 3: drawr - Initial screen with drawing sample

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 or Opera. 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 [1] 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 4) is defined in the app.js file which is used every time a new event listener and handler has to be defined.

```
89
90  function addEvent( element, event, callback ) {
91      if( window.addEventListener ) {
92          element.addEventListener( event, callback, false );
93      } else if( document.attachEvent ) {
94          element.attachEvent( 'on' + event, callback );
95      } else {
96          element[ 'on' + event ] = callback;
97      }
98  }
```

Fig. 4: A simple facade that masks the various browser specific methods

Here "callback" is the event handler function that 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 5, event listeners for individual elements (#paintsurface, #erase, etc.) are handled in same manner as done for component with collection of list elements (#palette).

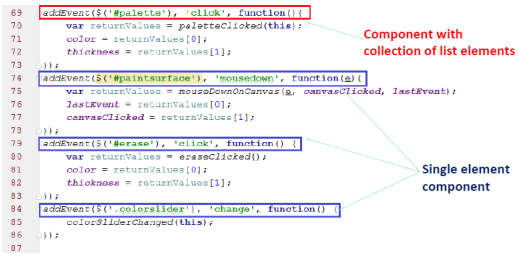

Fig. 5: Same interface of addEvent provided to all kinds of element components for adding event listeners

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 6.

Fig. 6: addEvent called recursively for each list item of composite component

4 CONCLUSION

———— Conclusion ————

REFERENCES

[1]flamingveggies. https://github.com/flamingveggies/drawr, 2015.

[2]Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.