Structural Patterns

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

Associate Editor: Guy Yachdav

ABSTRACT
JavaScript is one of the most popular and rapidly developing programming languages nowadays. Therefore good coding habits in JavaScript become an important issue, considering that more and more systems use this technology. In this seminar topic the usage of general patterns and avoiding anti-patterns was studied.

1 INTRODUCTION

A pattern in general is a reusable solution that can be applied to commonly occurring problems in the process of development - in this case - in developing JavaScript-powered programs. Although there are various frameworks and libraries built upon the JavaScript language, this papers topic does not consider any particular of them. General patterns are those which can be used in any JavaScript project regardless of the application context.

2 APPROACH

In order to demonstrate the application of general JavaScript patterns several projects on GitHub have been reviewed (see Section 6). In each project some code parts have been optimized by applying one or multiple patterns or by getting rid of anti-patterns.

3 STRUCTURAL PATTERNS

There are multiple universal patterns that can be applied in any JavaScript project. In this paper the following (anti-)patterns are reviewed [[1]]:

- *Function declarations* - creating anonymous functions and assigning them to a variable
- *Conditionals* - patterns and anti-pattern of using if else
- *Access to the global object* - accessing the global object without hard-coding the identifier window
- *Single var pattern* - one var statement for declaring multiple variables
- *Hoisting* - moving declarations to the top of the scope
- *For loops* - optimizing for loops
- *(Not) Augmenting built-in prototypes* - anti-pattern for introducing new functionality
- *Switch pattern* - pattern for using switch-case statements correctly
- *Implied typecasting* - anti-pattern for comparing objects
- *Avoiding eval()*
- *Number conversions with parseInt()* - pattern for using parseInt function
- *Avoiding setInterval()*

All of the (anti-)patterns from the list above are presented in this paper. However, not all of them were found in the refactored projects from GitHub (see Section 6). This section is going to cover the anti-patterns and suitable places to apply patterns that were encountered in the reviewed GitHub projects.

3.1 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 [1]. Facade pattern hides the complexities of the system and by providing an interface to the client using which the client can access the system.

3.1.1 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.

3.1.2 Structure Typical structure of Facade usage can be depicted as shown in the figure 1.

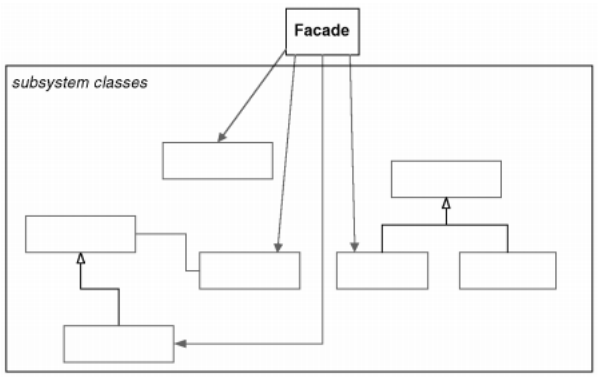

Fig. 1. Generic structure of the Facade pattern [1]

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.

3.1.3 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.

3.2 Composite Pattern

Conditionals are the ability to test a variable against a value and act appropriately in case the condition is met by the variable or not. They are also commonly called if-else statements by programmers.

```
if (type === 'foo' || type === 'bar') {
    ...
}
```

As stated [**?**] the if statement above (normal pattern) is only an antipattern when optimizing for low-bandwidth source (such as for a bookmarklet).
In this case one alternative is using regular expressions:

```
if (/^(foo|bar)\$/.test(type)) {
...
}
```

But using the normal pattern will generally outperform the regular expression in a loop.

Another alternative for low-bandwidth cases is the object literal:

```
if ({foo:1, bar:1}[type]) {
...
}
```

But the normal pattern is faster than the object literal for lower numbers of conditions, they generally even out around 10 conditions [**?**].

4 REFACTORING EXAMPLES

4.1 songFinder

4.2 drawr-bootstrap

5 CONCLUSION

Nowadays JavaScript is a very common language and a high percentage of the companies include Javascript in their projects. Therefore it is almost a must for any programmer to know the basics and concepts of JavaScript. General Patterns described in this paper help producing readable and maintainable source code.
This should be a major goal while writing any JavaScript program, since the projects containing JavaScript are written/read by more than one programmer. Therefore the efficiency of multiple programmers is increased, as they are used to the same patterns and styles while writing code.
In several cases - as for example loops - even the performance can be increased. [**?**] Therefore, those patterns can be considered as guidelines for any JavaScript program.

6 FURTHER READING

These general patterns appear in many projects, even though they are simple to realise. To show this - in addition to writing this article - the authors refactored four open source projects regarding general patterns. Those projects are listed below:

- CAC-Visit-Location *[**?**]*
- Stockwatch *[**?**]*
- Webgl_tumblr *[**?**]*

- Gist-embed *[**?**]*

The refactored files can be seen in the authors GitHub project: https://github.com/alex-vo/seminarJS.git

REFERENCES

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