How does the implementation of Strategy Design pattern in JavaScript affect Maintainability as measured by Cyclomatic complexity and Lines of code

Oskar Therén
Computer Engineering
The Institute of Technology at Linköping University
Email: oskth878@student.liu.se

***Abstract*—In this paper an implementation of the Strategy Design pattern, as described in the book Design Patterns: Elements of Reusable Object-Oriented Software, is presented in the high-level and untyped programming language JavaScript. It is implemented in an existing game, where it replaces a switch statement to reduce cyclomatic complexity. The difference in cyclomatic complexity and lines of code before and after the implementation is evaluated in the aspect of how it affects maintainability. This paper is concluded with some discussion about the usefulness of implementing the pattern in a bigger project.**

I. Introduction

It is widely acknowledged among object-oriented programmers that Design patterns can be useful to solve commonly occurring problems within a given context by creating reusable code. Design patterns was mainly introduced to software developers through the GoF (Gang of Four) that wrote about the concept in their book, Design Patterns: Elements of Reusable Object-Oriented Software [1], though the idea originally comes from an architectural concept. The patterns has since been widely discussed, customized and used.

Even though they are so popular there are few empirically justified reasons to use them according to C. Zhang and D. Budgen in their article "What Do We Know about the Effectiveness of Software Design Patterns?" [2].

The Design patterns are created from an object oriented perspective, while on the other hand web development is most commonly conducted in the dynamic, untyped and interpreted language JavaScript. JavaScript is often criticized by developers that are used to strongly typed languages, claiming that its static type-checking can lead to unsafe code. A solution to this, proposed by Harmes et al. [7], is to adapt the GoFs Design patterns to make them applicable to JavaScript, in an attempt to make the code both safer and more reusable.

The purpose of this paper is to find out if the implementation of a specific Design pattern, Strategy [Section I-B], will affect maintainability [Section I-D] in a positive way in the programming language JavaScript.

A. Research Question

- How does the implementation of Strategy Design pattern in JavaScript affect Maintainability as measured by Cyclomatic complexity and Lines of code

B. Strategy Design pattern

An UML-diagram of the concept is shown in [Figure 1].

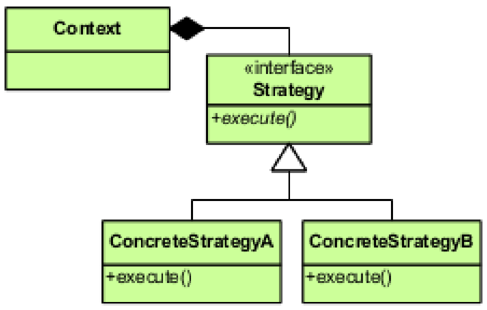

Fig. 1. Strategy pattern in a UML-diagram

Strategy is a behavioral pattern that intends to minimize coupling by encapsulating a family of algorithms and make them interchangeable by abstracting away the algorithms different functionalities into separate classes that implements a common interface, which is called strategy. For a language to be able to implement Strategy in the way it is explained by the GoF it needs something similar to an Java interface. This does not JavaScript have, which is further discussed in [Section II-A]. [1]

The GoF claims that one of the applications of the pattern is when "a class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class" [1]. Which gives motivation to why the example [Listing 1] in this paper is a good starting point.

C. SOLID

The purpose of many Design patterns is to solve some of the SOLID-violations that can arise when working on bigger projects. SOLID is an mnemonic acronym that stands for

- *Single responsibility*
  Each class should only have responsibility over one part of the softwares functionality.
- *Open-closed*
  Classes, functions and so on, should be open for extension but closed for modification.
- *Liskov substitution*
  Every subtype should be able to replace its inherited type.
- *Interface segregation*
  Clients should not be forced to implement methods from and interface that it will not use.
- *Dependency inversion*
  High level modules should not be affected by changes of low level modules.

These principles were developed by RC Martin in his books about Agile development but they can be applicable to a lot of different software development. They are generally good to follow when developing software and are all contributing to code that is for example easier to maintain. [3]

The Strategy pattern mainly improves the code with Single responsibility and Open-closed in aspect, although it follows the other principles as well.

Single Responsibility through having the code for the algorithms in separate classes.

Open-closed because when a new algorithm is introduced, the existing code will not have to be changed. Though a new class for that algorithm needs to be added so the interface is extended.

D. Maintainability

The total cost of maintenance in software development is widely discussed and different eminent names in software development have claimed that it will take up from 40 even up to 60 percent of the time and cost to maintain a project. In the 1990s it was claimed by two experts, Corbi and Yourdon, that software maintainability where going to be one of the major challenges for the 1990s. During the 90s this was confirmed by Hewlett-Packard that claimed that "they had between 40 and 50 million lines of code under maintenance and that 60 to 80 percent of research and development personnel are involved in maintenance activities" [4].

E. Metrics

Since the 1991s when the ISO standard for maintainability was introduced (ISO 9126) there have been several attempts to link maintainability with different metrics. Heitlager et al. argues that the problem when the standard was introduced is that "In general, the proposed metrics for assessing the maintainability characteristics are not measured on the subject of maintenance, i.e. the systems source code and documentation, but on the performance of the maintenance activity by the technical staff." [5]

There are several metrics that tries to measure the complexity of a function or a program. These metrics have different advancement levels, they range from easily measured metrics like just lines of code or cyclomatic complexity to Robillards interconnectivity metric that "integrates the structural as well as the textual aspects of a program in such a way that the organization of a program can be seen graphically. The measure of complexity depends on how a statement is related to the rest of the program" [6].

In the article Using Metrics to Evaluate Software System Maintainability Coleman et al. found that when they conducted automated software maintainability analysis on 11 software systems. They all corresponded to the experts intuition and also provided additional useful data. The metric they created is widely used and called MI (Maintainability Index) which is calculated as a factored formula consisting of Source Lines Of Code, Cyclomatic Complexity, Halstead volume and percent of lines of Comments. [4]

Heitlage et al. discusses several problems with MI they are summarized below:

- *Root-cause analysis*
  Since the formula just gives a number it can be hard to know what the cause of a bad value is.
- *Average complexity*
  The value will be small even though some functions might have high complexity, since most will not. They argue that an average value is misleading.
- *Computability*
  The Halstead Volume is difficult to define and compute.
- *Comment*
  Comments have no correlation with maintainability at all in general.
- *Understandability*
  The formula is hard to understand since it for example contains several constants that comes without any logical arguments.
- *Control*
  The developers and management can easily feel a lack of control over the MI value.

They moves on to suggest a metric of their own, though it is a bit more complex to go in depth on here. What is important for this paper is that they are also using Lines of Code and Cyclomatic Complexity in their analysis. Though they argues that it is better to calculate Cyclomatic Complexity per unit, where a unit is the smallest piece of code that can be executed and tested individually. Eg. a method in Java. [5]

II. Method

This section presents the JavaScript implementation of the Strategy pattern as well as the chosen metric to evaluate the solution.

A. Interpretation of the Strategy pattern for JavaScript

Since JavaScript is not an object oriented language the concept of interface does not exist. In Harmes et al. book "Pro JavaScript$^{\text{TM}}$ Design Patterns" [7] the recommendation is to create a Duck Typed interface emulation. This would be useful if the Strategy pattern where used in a real life program to ensure correct parameters where sent to the functions. But since this papers purpose is to evaluate Strategy pattern and uses a quite small example and JavaScript is additionally untyped, the implementation of an interface is skipped and type correctness is assumed.

The example used in this paper is taken from an existing game. The code that is supposed to be replaced with the Strategy pattern is a switch case that sets a `message` dependent on a string. It is part of a method that renders a message in the `MainUI` from the method parameter `entity`, see [Listing 1].

The switch case is a violation of the Open/closed principle the goal is to replace it with the calls in [Listing 2]. This is similar to how the call would look like in the object oriented language Java, a difference is that since JavaScript is untyped everything instantiated with `var`. The property `messageType` is replaced with what a more appropriate name `messageStrategy` and shall be a reference to the correct function instead of a string, e.g.

```
// New
entity.messageStrategy = new
    humanStrategy();
// Old
entity.messageType = "HumanMessage";
```

The Strategy pattern will be implemented in a separate file called `Messager.js`. It contains a prototype `Messager(strategy)` that takes a strategy and saves it to its own context `this`. It also contains a method `getMessage(...)` that calls the method `getMessage(...)` on the strategy bound to the context of `Messager`. The different strategies are in this case also placed in the file `Messager.js`, so when a new one is needed is just needs to be added to this file. For the resulting file see [Listing 3].

B. *Metrics*

The chosen metrics for this paper are cyclomatic complexity and lines of code, they are quite easy to measure and is applicable to many languages. Many maintainability metrics use a combination of these metrics in addition of some more.

1) Cyclomatic complexity: It was first developed by Thomas J. McCabe in his "A Complexity Measure" [10]. For this smaller example it can be summarized to how many paths a program has where each switch case and if statement creates a new path.

2) Lines of code: This metric can be used to predict maintainability of code, in general the amount of lines of code correlate with how maintainable the source code is. In this paper physical lines of code is used, that counts every line that is not whitespace or comment.

III. Result

The result of the two metrics are presented in the subsections below.

A. *Cyclomatic complexity*

The cyclomatic complexity of the original code is three, one for each case in the switch case.

After the implementation of the Strategy pattern the complexity is reduced down to one.

B. *Lines of code*

The number of lines of code in the original code is 18.

After the implementation of the Strategy pattern the total number of lines of code is increased to 29, though `MainUI.js` is reduced down to 5 lines.

IV. Discussion

The result and method is discussed in the following subsections.

A. *Result*

The cyclomatic complexity was completely removed from the method `showMessage` which was the expected result comparing with code examples found at two websites, though they were in the object oriented languages.

One example is by Gil Fink, that has among other co-authored several Microsoft Official Courses. He writes in a blog post about "Applying Strategy Pattern Instead of Using Switch Statements" where he applies the Strategy pattern to C++. [8]

Another example though from an unknown source, but. It is a similar implementation of the Strategy pattern made in C#. [9]

Lines of code had an increase which is to be expected. This can have a negative impact on maintainability, but since the increase of lines is in a new file this does in fact make the `MainUI.js` more maintainable and less coupled.

When the metrics are combined some conclusions can be drawn:

- No matter how big the switch case is `MainUI.js` will always have the same length with strategy and cyclomatic complexity will be reduced down to one.
- The total amount of line of code will increase with two for each switch case that is removed.

B. *Method*

The implementation of the pattern was not trivial to translate. Since JavaScript is untyped and does not have inheritance this implementation might seem a bit strange for some one not used to Design patterns and object oriented languages. Though if the project is done in object oriented inspired Design, this implementation is quite similar to the e.g. Java version, but then either some version of inheritance should be implemented or at least some check in `Messager.js` or in `showMessage()` to handle errors or a default behavior.

The metrics used are the two of the components that Visual Studio by Microsoft uses [11]. The conclusions drawn from this paper are that they seem to work equally well in JavaScript compared to object oriented languages.

V. Conclusion

The definition of the Strategy pattern from the GoF is in some ways lost when translated into JavaScript. Interfaces does not exist and since it is untyped there would be no purpose using inheritance for this pattern. Though if the strategy pattern would be used reoccurring in a project the Duck Typed interface implementation by Harmes et al. is recommended to gain robuster code. [7]

The SOLID principles can be beneficial to implement in JavaScript if the project tries to do a object oriented inspired Design. Design pattern such as Strategy pattern can be implemented quite similarly as shown in this paper and gain some of the benefits to Single responsibility and Open-closed principals.

Despite the benefits, this paper shows that the implementation can increase total lines of code which may have a maintainability drawback. And in search for a in depth evaluation of the pattern in JavaScript the results where only examples of different implementations in JavaScript and explanations why the pattern is beneficial in general object oriented Design.

Code

```
// Creates and shows a message dependent
    on entity.messageType
MainUI.prototype.showMessage = function(
    entity) {
  var message;
  switch(entity.messageType) {
    case "HumanMessage":
      ...
      message = ...
      break;
    case "ZombieMessage":
      ...
      message = ...
      break;
    case "MonsterMessage":
      ...
      message = ...
      break;
  }
  ...
};
```

Listing 1. `MainUI.js` The original switch case.

```
// Creates and shows a message dependent
    on entity.messageStategy
MainUI.prototype.showMessage = function(
    entity) {
  var messager = new Messager(entity.
    messageStrategy);
  var message = messager.getMessage(...);
  ...
};
```

Listing 2. `MainUI.js` Switch case replaced trough the Strategy pattern.

```
Messager = function(strategy) {
  this.strategy = strategy;
};

Messager.prototype.getMessage = function
    (...) {
  return this.strategy.getMessage(...);
};

var humanStrategy = function() {
  this.getMessage = function(...) {
    ...
    return message;
  };
};

var zombieStrategy = function() {
  this.getMessage = function(...) {
    ...
    return message;
  };
};

var monsterStrategy = function() {
  this.getMessage = function(...) {
    ...
    return message;
  };
};
```

Listing 3. `Messenger.js` Resultant Strategy code strategy function names are the values of `ObjectTypeEnum`.

References

[1] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* Addison-Wesley Professional, 1st edition, January 15, 1995

[2] Cheng Zhang and David Budgen, *What Do We Know about the Effectiveness of Software Design Patterns?* IEEE Transactions on Software Engineering, vol. 38, no. 5, pp. 1213-1231, September-October 2012

[3] Robert C. Martin, *Agile software development: principles, patterns, and practices* Agile software development: principles, patterns, and practices, 2002

[4] Don Coleman, Dan Ash, Bruce Lowther and Paul Oman, *Using Metrics to Evaluate Software System Maintainability* Computer, Volume:27, Issue: 8, August 1994

[5] Heitlager, Ilja, Tobias Kuipers, and Joost Visser. *A practical model for measuring maintainability* In Quality of Information and Communications Technology, 2007, pages 30-39

[6] Pierre N. Robillard and Germinal Boloix, *The Interconnectivity Metrics: A New Metric Showing How a Program Is Organized* Journal of Systems and Software, Volume 10, Issue 1, July 1989, Pages 29-39

[7] Dustin Diaz and Ross Harmes, *Pro JavaScript design patterns* Apress, 2008

[8] Gil Fink, *Applying Strategy Pattern Instead of Using Switch Statements*, http://blogs.microsoft.co.il/gilf/2009/11/22/applying-strategy-pattern-instead-of-using-switch-statements/, Accessed: 2017-06-07

[9] *How to use Strategy Pattern Instead of Switch-Case statements*, https://vcpptips.wordpress.com/tag/cyclomatic-complexity/, Accessed: 2017-06-07

[10] Thomas J. McCabe, *A Complexity Measure* in IEEE Transactions on Software Engineering, vol. SE-2, no. 4, pp. 308-320, Dec. 1976

[11] Microsoft Developer Network, *Code Metrics Values*, https://msdn.microsoft.com/en-us/library/bb385914.aspx, Accessed: 2017-06-08

Improvements after submission