How does the implementation of Strategy Design Pattern in JavaScript affect Maintainability as measured by Maintainability Index

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 originally 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 the codes Cyclomatic Complexity. The difference before and after the implementation is evaluated with the help of the metric Maintainability Index 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

Design Patterns were 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. It is widely acknowledged in object-oriented programming that Design Patterns can be useful for solving commonly occurring problems within a given context by creating reusable code. Today patterns are often discussed, customized and used in a big part of software development.

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. More and more businesses are moving their products over to the web and with that comes high expectations on code quality. Harmes et al. [2] and Osmani [3] have proposed to adapt the GoF's Design Patterns to make them applicable to JavaScript, in an attempt to make the code safer, more reusable and easier to maintain.

When using Design Patterns it might not always be good to follow them blindly. Zhang et al. argues in their article "What Do We Know about the Effectiveness of Software Design Patterns?" that even though Design Patterns are so popular, there are few empirically justified reasons to use them. [4]

The purpose of this paper is to find out if the implementation of a specific Design Pattern, will affect maintainability in a positive way, measured by a metric in the programming language JavaScript. All of these concepts are in depth explained in the Background of this paper Section II.

A. Research Question

- How does the implementation of Strategy Design Pattern in JavaScript affect Maintainability as measured by Maintainability Index?

II. Background

A. Strategy Design Pattern

Strategy is a behavioral pattern that is created 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. An UML diagram of the concept is shown in Figure 1. For a language to be able to implement strategy in the way it is explained by the GoF it needs something similar to a Java interface. JavaScript does not have the concept of interface, which is further discussed in Section III-A. [1]

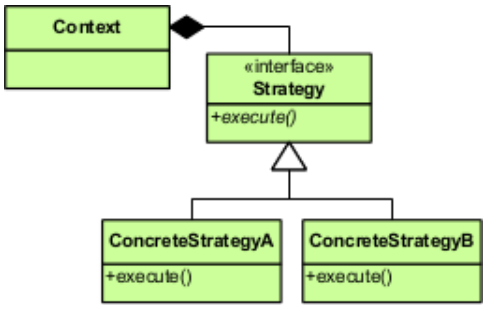

Figure 1: Strategy Pattern in a UML diagram.

The GoF claims that one of the main applications of the pattern is to put conditional branches into new classes they call strategy classes when there several conditionals with many behaviors [1]. This motivates that the example used in this paper, Listing 1, is a good starting point.

When discussing Design Patterns it is important to note that they are not always the perfect solution and have received

some criticism. Zhang et al. is critical to the fact that some Design Patterns benefits are supported by weak empirical evidence. [4]

"The Difficulties of Using Design Patterns among Novices: An Exploratory Study" by Abdul Jalil et al. explores novice programmers' difficulties when applying Design Patterns. Their conclusion on the Strategy Pattern is that it is not so hard to understand nor apply. This study says nothing about the actual effect on the code. [5]

Khomh et al. wanted to empirically evaluate how some software qualities are affected by design patterns. They showed that some patterns generally have negative impact on some quality attributes. The Strategy Pattern did according to their study have a positive impact on some aspects such as simplicity, modularity, learnability and understandability, all important attributes relating to maintainability. On the other hand they showed results of negative impact on reusability and robustness. [6]

B. 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 software's 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 an 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 aspects of software development. They are generally good to follow when developing software and are all contributing to code that is for example easier to maintain. [7]

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 to extend the interface.

C. Design Patterns in JavaScript

There are some attempts of using Design Patterns in JavaScript, with different levels of modification. In the book "Pro JavaScript™ Design Patterns" from Harmes et al. they try to create GoF's book for JavaScript developers which they motivates with the following reasons:

- *Maintainability*
  Makes modules less coupled which makes refactoring easier.
- *Communication*
  Easier to discuss the code on a higher level with other developers.
- *Performance*
  Some patterns can improve the performance of the program.

They also mention some drawbacks with Design Patterns, some of them can have a negative performance impact while others can make the code more complex and harder to understand for newer developers. [2]

Another book on the subject is Osmanis "Learning JavaScript Design Patterns: A JavaScript and jQuery Developer's Guide" which provides his interpretation of some of the GoF's Design Patterns both for JavaScript and the library jQuery. He discusses the different patterns and their usefulness in JavaScript. [3]

Both books contains arguments to why Design Patterns can be useful in JavaScript but neither implements the Strategy Pattern. There is no obvious reason for this since the Strategy Pattern can be applied to remove the use of multiple conditionals, as mentioned in Section II-A, which is as relevant in JavaScript as in object oriented languages. That makes the pattern an interesting choice for this paper.

D. Maintainability and Metrics

Since the 1991 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 a problem with the standard is that the suggested maintainability metrics measure the wrong things. They focus on the technical staffs maintenance performance instead of measuring the systems source code. [8]

There are several metrics that try to measure the complexity of a function or a program. These metrics have different advancement levels, some are easily measured like Lines of Code. Another simple metric is Cyclomatic Complexity, which is the number of linearly independent paths through some code. The point of the metric is that many paths indicates that the code might both be hard to maintain and test. The metric was first developed by Thomas J. McCabe in his "A Complexity Measure". [9]

On the other side there are metrics like 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." [10]

In the paper "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 by among others Visual Studio by Microsoft [11]. It is 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. Where Halstead Volume is calculated from the programs length, amount of operands and amount of operators. [12]

According to Heitlage et al. there are several problems with MI:

1) Since the formula just gives a number it can be hard to know what the cause of a bad value is.
2) Average complexity will be small even though some methods might have high complexity, since most methods will not. They argue that an average value in this case is misleading.
3) The Halstead Volume is difficult to define and compute.
4) Comments have no correlation with maintainability at all in general. The formula is hard to understand since it for example contains several constants that are introduced without any logical arguments.
5) The developers and management can easily feel a lack of control over the MI value.

They move on to suggest a metric of their own which also takes Lines of Code and Cyclomatic Complexity into account. They argue 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. [8]

Gill et al. uses the name Complexity Density for the ratio between Lines of Code and Cyclomatic Complexity of each module. The goal of their report, "Cyclomatic Complexity Density and Software Maintenance Productivity", was to evaluate how their metric affects productivity of maintenance. The metric was tested on some projects and they claim that the ratio was a good predictor of maintenance costs of modules. [13]

III. Method

This section presents the JavaScript implementation of the Strategy Pattern as well as how the chosen metric is used to evaluate the solution.

A. Interpretation of the Strategy Pattern for JavaScript

Since JavaScript is not an object oriented language the concept of an interface does not exist. Harmes et al. recommend creating a Duck Typed interface emulation, which is to assume that an interface is implemented if a object contains all the methods the interface has. They recommend the use of a method that ensures that the interface is implemented. This would be useful if the Strategy Pattern was used in a real life program. But since this paper's purpose is to evaluate Strategy Pattern, it uses quite a small example and JavaScript is untyped, the implementation of an interface is skipped and type correctness is assumed. This choice is backed since Harmes et al. claims that the use of an interface becomes most beneficial when the system gets more complex. [2]

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 statement 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 statement 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, with a difference being that, since JavaScript is untyped everything is instantiated with `var`. The property `messageType` is replaced with 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 is 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 it just needs to be added to this file. For the resulting file see Listing 3.

B. Metric

The choice of metric is not trivial, as discussed in Section II-D. The choosen metric is MI since it is still broadly used by e.g. Microsoft, the comments by Heitlage et al. is discussed in Section V. Since MI is quite complex it will probably be hard to implement, as argued by Heitlage et al. [8]. Therefore a program, complexity-report [14], is used that output information about JavaScript code. Microsofts version of MI is used, where the only difference is that it does not take comments in account, as sugested by Heitlage et al., and gives a value between 0 and 100. It is calculated as follows:

```
Maintainability Index =
  MAX(0, (171-5.2 *ln(Halstead Volume)
             -0.23*Cyclomatic Complexity
             -16.2*ln(Lines of Code)
         )*100/171)
```

The different components are explained bellow.

1) Halstead Volume: Halstead Volume is calculated as follows

```
Halstead Volume =
  (the total number of operators +
   the total number of operands) *
```

```
log(the number of distinct operators +
    the number of distinct operands)
```

2) Cyclomatic Complexity: It can be summarized to how many paths a program has where each switch statement and if statement creates a new path.

3) Lines of Code: In MI, Logical Lines of Code are used, a count of every programming language statement.

IV. Result

The result from complexity-report is presented in Table 1. Additionally to MI, Lines of Code and Cyclomatic Complexity is also presented.

	Switch	Strategy
Mean per-function logical LOC	44	2.5
Highest function Cyclomatic Complexity	14	3
Mean per-function Cyclomatic Complexity	14	1.1
Mean per-module maintainability index	46	81

Table 1: The result from complexity-report on the two different examples.

V. Discussion

The result and method are discussed in the following subsections.

A. Result

The Cyclomatic Complexity per function and the highest value was reduced drastically, which was the expected result when 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++. [15]

Another example, though from an unknown source, is a similar implementation of the Strategy Pattern made in C#. [16]

Total Lines of Code had an increase but reduced per module, 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 looking at the Maintainability Index, there is an significant increase. If new cases needs to be added to the example the MI of the old code decreases faster than the Strategy example which is more stable. Worth noting is that according to Welker [17] MI is an effective approach, but the result is something that should not be followed blindly and taken as truth it merely gives a little view into the complex issues of maintenance.

Ampatzoglou et al. did an implementation of the strategy pattern among others in two games and they also concluded that it helps in reducing the complexity of the code. [18]

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 someone not used to Design Patterns and object oriented languages. Though if the project were to be done in object oriented inspired design, this implementation would be quite similar to e.g. the Java version, but then either some version of inheritance should be implemented or at least some check in `Messager.js` or in `showMessage()` for handling errors or defining a default behavior.

Addressing some of the issues mentioned in Section II-D:

1) When comparing two examples and the number is normalized between 0-100 the number still gives a relative difference which shows which solution is best.
2) Since I only calculated the MI of one the relevant portion of the source code this value should not be misleading.
3) The use of a tool like complexity-report is very helpful here.
4) In Microsofts version of MI comments are not used, and this example only had one comment.

The conclusions drawn from this paper on the metric is that it seem to work well in JavaScript though further investigation would be interesting.

VI. Conclusion

The definition of the Strategy Pattern from the GoF is in some ways lost when translated into JavaScript. Interfaces do not exist, and since JavaScript is untyped there would be no point in using inheritance for this pattern. Though if the Strategy Pattern would be used multiple times in a project, the Duck Typed interface implementation by Harmes et al. is recommended to more robust code. [2]

The SOLID principles can be beneficial to follow in JavaScript, especially if the developers tries to do an 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 from Single responsibility and Open-closed principals.

Strategy Pattern has other maintainability benefits shown by Balazinska1 et al. They used to reduce code cloning and at the same time reduce the coupling of the code base. [19]

Despite the great increase in maintainability according to the Maintainability Index, the result show that the implementation can increase total Lines of Code, which may have a maintainability drawback. When searching for an in depth evaluation of the pattern in JavaScript, the results consisted only of examples of different implementations in JavaScript and explanations of 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":
      if (Math.random() < 0.5)
        message += "Hello";
      else
        message += "Hi";
      break;
    case "ZombieMessage":
      message = entity.enemyPrefix;
      var rand = Math.random();
      if (rand < 0.33)
        message += "Brains";
      else if (rand < 0.67)
        message += "Uuuuh";
      else
        message += "Wouoo";
      break;
    case "MonsterMessage":
      message = entity.enemyPrefix + "
    Grrr";
      break;
    case "DogMessage":
      message = entity.enemyPrefix + "
    Woof";
      break;
    case "CatMessage":
      if (Math.random() < 0.5)
        message = "Meow";
      else
        message = "Hiss";
      break;
    case "CowMessage":
      message = "Muuu";
      break;
    case "CarMessage":
      message = "Wroom";
      break;
    case "VampireMessage":
      message = entity.enemyPrefix + "I
    want your blood";
      break;
    case "GoatMessage":
      message = "Baah"
      break;
  }
  renderMessage(message);
};
```

Listing 1: `MainUI.js` The original switch statement.

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

Listing 2: `MainUI.js` Switch statement 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(enemyPrefix)
     {
    var message;
    if (Math.random() < 0.5)
      message += "Hello";
    else
      message += "Hi";
    return message;
  };
};
var zombieStrategy = function() {
  this.getMessage = function(enemyPrefix)
     {
    var message = enemyPrefix;
    var rand = Math.random();
    if (rand < 0.33)
      message += "Brains";
    else if (rand < 0.67)
      message += "Uuuuh";
    else
      message += "Wouoo";
    return message;
  };
};
var monsterStrategy = function() {
  this.getMessage = function(enemyPrefix)
     {
    return enemyPrefix + "Grrr";
  };
};
var dogStrategy = function() {
  this.getMessage = function(enemyPrefix)
     {
    return enemyPrefix + "Woof";
  };
};
var catStrategy = function() {
  this.getMessage = function(enemyPrefix)
     {
    var message;
    if (Math.random() < 0.5)
      message = "Meow";
    else
      message = "Hiss";
    return message;
  };
};
var cowStrategy = function() {
  this.getMessage = function(enemyPrefix)
     {
    return "Muuu";
  };
};
var carStrategy = function() {
  this.getMessage = function(enemyPrefix)
     {
    return "Wroom";
  };
};
var vampireStrategy = function() {
  this.getMessage = function(enemyPrefix)
     {
    return enemyPrefix + "I want your
     blood";
  };
};
var goatStrategy = function() {
  this.getMessage = function(enemyPrefix)
     {
    return "Baah";
  };
};
```

Listing 3: `Messenger.js` Resultant strategy code. The 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 1995

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

[3] Osmani, Addy, *Learning JavaScript Design Patterns: A JavaScript and jQuery Developer's Guide* O'Reilly Media, Inc., 2012

[4] 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, pages 1213-1231, September-October 2012

[5] Jalil, Masita Abdul, and Shahrul Azman Mohd Noah, *The difficulties of using design patterns among novices: An exploratory study.* Computational Science and its Applications, International Conference on, pages 97-103, 2007

[6] Khomh, Foutse, Yann-Gal Guhneuc, and Ptidej Team, *An empirical study of design patterns and software quality* GEODESResearch Group on Open, Distributed Systems, Experimental Software Engineering, January 2008

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

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

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

[10] 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, Pages 29-39, July 1989

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

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

[13] Gill, Geoffrey K., and Chris F. Kemerer, *Cyclomatic complexity density and software maintenance productivity* IEEE transactions on software engineering 17.12, Pages 1284-1288, 1991

[14] Escomplex *complexity-report* https://www.npmjs.com/package/complexity-report, Accessed: 2018-05-27

[15] 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

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

[17] Welker KD., *The software maintainability index revisited* CrossTalk 14, Pages 18-21, August 2001

[18] Ampatzoglou A, Chatzigeorgiou A., *Evaluation of object-oriented design patterns in game development* Information and Software Technology, pages 445-454, May 2007

[19] Balazinska M, Merlo E, Dagenais M, Lague B, Kontogiannis K., *Partial redesign of Java software systems based on clone analysis* In Reverse Engineering, 1999. Proceedings. Sixth Working Conference, Pages 326-336, October 1999

Improvements after submission

A. Resubmission 1 (2017-09-09)

I have tried to rectify all the problems in your assessment. I have mainly done improvements to the first two sections of the paper, since most feedback was about sources and motivation and connection of the metric, JavaScript, maintainability.

I have also added several new sources and removed some old. I have tried to improve the language quality of the entire paper. The total length of the paper have been increased by circa 1000 words despite the fact that some old material have been removed.

B. Resubmission 2 (2018-01-04)

A lot of changes to sections I & II since most of the criticism was to those parts. I have tried to connect JS, the metrics and maintainability more. Removed some general parts about maintainability and added some more specific information, same with CC and LoC.

Also some minor overall text quality changes and some explanations about different concepts.

C. Resubmission 3 (2018-06-02)

Switched metric to MI, and therefore had to change a lot of Section III-B and the sections after that. Added four more sources.