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 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 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 in object-oriented programming that Design Patterns can be useful for solving commonly occurring problems within a given context by creating reusable code. 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. The patterns have since been widely discussed, customized and used.

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 dynamic type-checking can lead to unsafe code. A solution to this, proposed by Harmes et al. [2] and Osmani [3], is 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 two metrics 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 Cyclomatic Complexity and Lines of Code?

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 implement 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]

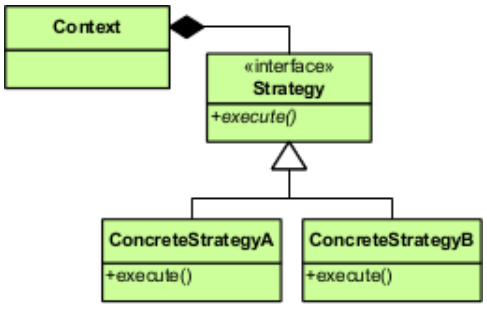

Fig. 1. Strategy Pattern in a UML-diagram.

The GoF claims that one of the main 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]. This is the motivation to why

the example used in this paper [Listing 1] is a good starting point.

When discussing Design Patterns it is important to note that is not always a 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. In the case of the Strategy Pattern Zhang claims that it is only mentioned in one study. [4]

The study they mention is "The Difficulties of Using Design Patterns among Novices: An Exploratory Study" by Abdul Jalil et al. which 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]

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 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 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. [6]

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 which are that some of them can have a negative performance impact while others can make the code more complex and harder to understand by 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 on 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 is an interesting choice for this paper.

D. Maintainability and Metrics

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 anywhere from 40 even up to 60 percent of the time and cost of maintaining 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. 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." [7]

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 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." [8]

There are several metrics that try to measure the complexity of a function or a program. These metrics have different advancement levels, they range from easily measured metrics like 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." [9]

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 [10]. 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. [7]

Heitlage et al. discusses several problems with MI, which 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*
  This value 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.
- *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 are introduced without any logical arguments.
- *Control*
  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, though it contains too many factors for an in depth explanation here. What is important for this paper is that they are also using Lines of Code and Cyclomatic Complexity in their analysis. Though 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]

III. Method

This section presents the JavaScript implementation of the Strategy Pattern as well as how the chosen metrics are 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. This would be useful if the Strategy Pattern was used in a real life program to ensure correct parameters were sent to the functions. But since this paper's purpose is to evaluate Strategy Pattern, uses quite a small example, and JavaScript is untyped, the implementation of an interface is skipped and type correctness is assumed. 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. Metrics

The choice of metrics is not trivial, as discussed in [Section II-D]. The preferred metric for a bigger project would be either MI or something similar to what is suggested by Heitlage et al. Since MI is quite complex it will probably be hard to implement and understand in this example, as argued by Heitlage et al. [8]. Implementing all the metrics suggested by Heitlage et al. would take quite long time, hence the chosen metrics for this paper being a subset from both Heitlage et al. and MI. Cyclomatic Complexity and Lines of Code are quite easy to measure and are applicable to many languages, they are explained bellow:

1) Cyclomatic Complexity: It was first developed by Thomas J. McCabe in his "A Complexity Measure" [11]. For this smaller example it can be summarized to how many paths a program has where each switch statement 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 are used, a count of every line that is not whitespace or comment.

IV. Result

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

A. Cyclomatic Complexity

The Cyclomatic Complexity of the original code is three, one for each case in the switch statement.

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.

V. Discussion

The result and method are discussed in the following subsections.

A. Result

The Cyclomatic Complexity was reduced down one (which is the lowest possible number) in the function `showMessage`, 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++. [12]

Another example, though from an unknown source, is a similar implementation of the Strategy Pattern made in C#. [13]

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 statement 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 statement 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 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.

The conclusions drawn from this paper on the metrics are that they seem to work equally well in JavaScript as they do in oriented languages.

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.

Despite the benefits, 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":
      ...
      message = ...
      break;
    case "ZombieMessage":
      ...
      message = ...
      break;
    case "MonsterMessage":
      ...
      message = ...
      break;
  }
  ...
};
```

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(...);
  ...
};
```

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(...) {
    ...
    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. 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 15, 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] Robert C. Martin, *Agile software development: principles, patterns, and practices* Agile software development: principles, patterns, and practices, 2002

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

[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] 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

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

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

[12] 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

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

IMPROVEMENTS AFTER SUBMISSION

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.