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 report an implementation of the Prototype design pattern, as described by the GoF, is presented in the high-level and untyped programming language JavaScript. It is implemented into the context of an existing game, where it replaces a switch case to remove cyclomatic complexity. The cyclomatic complexity difference is compared to the difference in lines of code to evaluate how maintainability is affected.

I. INTRODUCTION

It is widely acknowledged among object-oriented programmers that design patterns are useful to solve commonly occurring problems within a given context when coding. Design patterns was mainly introduced to software developers through the GoF (Gang of Four) that came out with the idea in their book from 1995 [1] the idea originaly came from an architectural concept. The patterns has since been widely discussed 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 purpose of this report 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.

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

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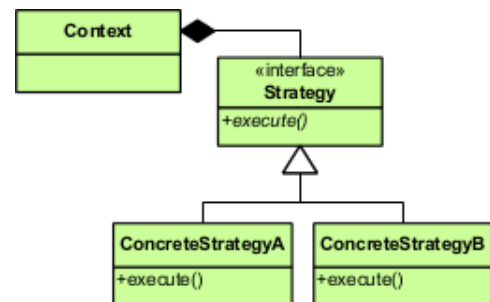

Fig. 1. Strategy pattern in a UML-diagram

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.

All of these principles are good to follow when developing software and are all contributing to code that is easier to maintain [3].

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 since adding a new algorithm the other code will not have to be changed, just add a new class for that algorithm and extend the interface.

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

There are several metrics that tries to measure the complexity of a function or a program. There are a lot of different advancement levels of these metrics, they can 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" [5].

Since the 1990s there have been several attempts to link maintainability with different metrics. In the article Using Metrics to Evaluate Software System Maintainability they found that when they conducted automated software maintainability analysis on 11 softwaresystems. They all corresponded to the experts intuition and also provided additional useful data [4].

Wei and Henry came to similar conclusion when studing Object-Oriented Metrics that Predict Maintainability [6].

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™ 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 reports 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 report 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 report are cyclomatic complexity and lines of code, they are quite easy to measure and is applicable to many languages.

1) Cyclomatic complexity: id anad switch....

2) Lines of code: The higher amount of lines of code the worse maintainability, excluding whitespace and comments....

III. RESULT

The result of the two metrics are presented 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 (excluding whitespace), though `MainUI.js` is reduced down to 5 lines.

IV. DISCUSSION

The result and method is discussed in the following subsections.

A. Method

Something something

B. Result

The cyclomatic complexity was completely removed from the function `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]

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

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 report and gain some of the benefits to Single responsibility and Open-closed principals.

Despite the benefits, this report 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.

```
// 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, *Principles Of OOD*, http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod, Accessed: 2015-10-17, Dates back to at least 2003

[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] 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

[6] Wei Li and Sallie Henry, *Object-Oriented Metrics that Predict Maintainability*, Journal of Systems and Software, Volume 23, Issue 2, November 1993, Pages 111-122

[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