# 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*—**

## 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 Gang of Four that came out with the idea in their book from 1995 [1] the idea originaly came from an architectural concept. and since then they have 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

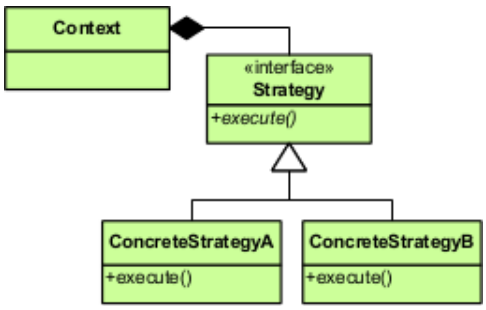


Fig. 1. Strategy pattern in a UML-diagram

Strategy is a behavioral pattern that intends to handle polymorphism 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]

### 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* trough when 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 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" [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

safa

### A. Interpretation of the Strategy pattern for JavaScript

Firstly, 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 since JavaScript is loosely typed 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 see [Listing 1].

The switch case is a violation of the Open/closed principle the goal is to replace this code 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 loosely typed everything instantiated with `var`. `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. `entity.messageStrategy = new humanStrategy()` instead of `entity.messageType = "HumanMessage"`.

The Strategy pattern will be implemented in a separate file called `Messager.js`. `Messager(strategy)` will be its main function that takes a strategy and saves it to its own context `this`. It also contains a function `getMessage(...)` that calls the function `getMessage(...)` on the strategy bound to `this`. `Messager.js` also contains all the different strategies, so when a new one is needed is just needs to be added to this 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.

## 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 15.

After the implementation of the Strategy pattern the total number of lines of code is increased to 30, though `MainUI.js` is reduced down to 4 lines.

## IV. Discussion

The result and method is discussed in the following subsections.

### A. Method

### B. Result

The cyclomatic complexity was completely removed from the function `showMessage`

## V. Conclusion

The conclusion goes here.

## APPENDIX

```
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.

```
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