

**Department of Computing**

**Te Horo Rorohiko**

**CH3880 Bachelor of Information and Communications Technology**

**BCCE301 Cooperative Education Project**

**Semester 1, 2017**

**DEVELOPING JAVASCRIPT WEB APPLICATION FOR MOODLE 3**

**Ara Institute of Canterbury**

**METHODOLOGY ESSAY**

**ON REFACTORING**

Gervis Chen

99117371

Word Count: 3395

### Table of Contents

[Table of Contents 1](#_Toc486846868)

[Introduction 3](#_Toc486846869)

[Part A - Theory - Refactoring 3](#_Toc486846870)

[Why Refactoring? 3](#_Toc486846871)

[History 3](#_Toc486846872)

[What is Refactoring? 3](#_Toc486846873)

[List of Code smells 4](#_Toc486846874)

[Pros and cons 6](#_Toc486846875)

[When to use refactoring 6](#_Toc486846876)

[When not to use refactoring 6](#_Toc486846877)

[How to apply 7](#_Toc486846878)

[Part B - Industry work 7](#_Toc486846879)

[Why I chose Refactoring? 7](#_Toc486846880)

[How I applied Refactoring 8](#_Toc486846881)

[Additional practices 8](#_Toc486846882)

[A specific example 9](#_Toc486846883)

[Before 10](#_Toc486846884)

[After 11](#_Toc486846885)

[Refactoring in the industry 12](#_Toc486846886)

[Part C - Compare and contrast 13](#_Toc486846887)

[Fowler 13](#_Toc486846888)

[Code smells 13](#_Toc486846889)

[Tools 13](#_Toc486846890)

[Usefulness 14](#_Toc486846891)

[Conclusion 14](#_Toc486846892)

[References 15](#_Toc486846893)

Figure 1 Red-green refactor cycle 7

Figure 2 Refactoring cycle 8

Figure 3 Tests passing 10

Table 1 Fowler’s (1999) Code smells 4

Table 2 Before code 9

Table 3 Refactoring steps 11

Table 4 After code 12

### Introduction

In this report, the methodology used in the project is refactoring. The report will be split into three sections: 1) the theory of refactoring, 2) the processes involved with the application of refactoring in the project and industry, and 3) a compare and contrast of refactoring used in the project with industry and theory.

# Part A - Theory – Agile Project Management

#### Why ?

When a company wants to full the extreme program in the transition and make the use of program beyond the project development team, it needs to face some problems.

A bad communication between clients and developers.

Misunderstanding of user stories, the plan, and velocity.

A blurry deliver credible plans and schedules

So Be agile in adapting to client feedback and changed requirements is very important for software management.

#### History

Agile management of IT project is widely used for software development. In the earlier stages of development the norm was to use “heavy” methodologies such as Waterfall or Spiral for software development. A common agile method, Extreme Programing, was developed in the mid 1990’s, but at that time there were not yet any guidelines that detailed what Agile was. In 2001 seventeen software developers meet in Utah to discuss the new ways of managing software projects and published the Agile Manifesto for Software Development.(The Agile Alliance). The Manifesto underlined common values and principals of all agile projects and led to what Agile is today. Agile Project Management is a lightweight method for software development projects.

#### What is Extreme Programing?

|  |  |
| --- | --- |
| The first Extreme Programming project was started March 6, 1996. Extreme Programming is one of several popular [Agile Processes](http://www.agile-process.org/). It has already been proven to be very successful at many companies of all different sizes and industries world wide. http://www.extremeprogramming.org/images/pixel.gifExtreme Programming is successful because it stresses customer satisfaction. Instead of delivering everything you could possibly want on some date far in the future this process delivers the software you need as you need it. Extreme Programming empowers your developers to confidently respond to changing customer requirements, even late in the life cycle. http://www.extremeprogramming.org/images/pixel.gifExtreme Programming emphasizes teamwork. Managers, customers, and developers are all equal partners in a collaborative team. Extreme Programming implements a simple, yet effective environment enabling teams to become highly productive. The team self-organizes around the problem to solve it as efficiently as possible.  http://www.extremeprogramming.org/images/pixel.gifExtreme Programming improves a software project in five essential ways; communication, simplicity, feedback, respect, and courage. Extreme Programmers constantly communicate with their customers and fellow programmers. They keep their design simple and clean. They get feedback by testing their software starting on day one. They deliver the system to the customers as early as possible and implement changes as suggested. Every small success deepens their respect for the unique contributions of each and every team member. With this foundation Extreme Programmers are able to courageously respond to changing requirements and technology. http://www.extremeprogramming.org/images/pixel.gifThe most surprising aspect of Extreme Programming is its [simple rules](http://www.extremeprogramming.org/rules.html). Extreme Programming is a lot like a jig saw puzzle. There are many small pieces. Individually the pieceshttp://www.extremeprogramming.org/images/pixel.gif | [gile flow chart](http://www.agile-process.org/) make no sense, but when combined together a complete picture can be seen. The rules may seem awkward and perhaps even naive at first, but are based on sound [values](http://www.extremeprogramming.org/values.html) and principles. http://www.extremeprogramming.org/images/pixel.gifOur rules set expectations between team members but are not the end goal themselves. You will come to realize these rules define an environment that promotes team collaboration and empowerment, that is your goal. Once achieved productive teamwork will continue even as rules are changed to fit your company's specific needs. http://www.extremeprogramming.org/images/pixel.gifThis [flow chart](http://www.extremeprogramming.org/map/project.html) shows how Extreme Programming's rules work together. Customers enjoy being partners in the software process, developers actively contribute regardless of experience level, and managers concentrate on communication and relationships. Unproductive activities have been trimmed to reduce costs and frustration of everyone involved. |

#### List of Code smells

There are refactoring techniques that helps to remove code smells. The code smell is generally indicative of which technique to use and the context is looked at to identify any other techniques to use. Usually, refactoring involves having to use multiple techniques to remove the code smell. The following table lists all of Fowler’s (1999) code smells:

Table 1 Fowler’s (1999) Code smells

|  |  |  |
| --- | --- | --- |
| **Category** | **Code smell** | **Description** |
| Bloaters | Long method | A method that contains too many lines of code |
|  | Large class | A class that contains too many lines of code or handles too many functions. |
|  | Primitive obsession | Excessive use of primitive data type (strings, integers etc.) in fields or arrays for storing data instead of classes. |
|  | Long parameter list | A method with a long parameter list. |
|  | Data clumps | Identical groups of variables used in other parts of the code. |

|  |  |  |
| --- | --- | --- |
| **Category** | **Code smell** | **Description** |
| Change preventers | Divergent change | When a change is made, changes have to be made to a single class. |
|  | Shotgun surgery | When a change is made, changes have to be made to multiple classes. |
|  | Parallel inheritance hierarchies | When a subclass is created for one class, a subclass has to be created for another. |
| Object-orientation abusers | Switch statements | Large “switch” and “if” statements |
|  | Temporary field | Fields are used only when needed by objects and when they are unused, there they are empty. |
|  | Refused bequest | A subclass of a superclass that uses some of its inherited methods and fields while the others are never used. |
| Dispensables | Comments | There are comments that explain a particular line of code or method, etc. |
|  | Duplicate code | Two or more lines of code that looks the same or very similar. |
|  | Lazy class | A class that has little or no purpose. |
|  | Data class | A class that only contains data and accessor methods. |
|  | Dead code | Code that has become unused |
|  | Speculative generality | Unused class, method, field, or parameter |
| Couplers | Feature envy | A method that uses the data of another class more than its own. |
|  | Inappropriate intimacy | A class that uses fields and methods of another class more than its own. |
|  | Message chains | A chain of method calls. |
|  | Middle man | A class that has only one purpose, which is to pass parameters to methods of other classes. |

#### Pros and cons

The pros and cons of refactoring are listed in this section.

* One advantage of this approach is allowing for cost estimates-based software features instead of developer activity. This allows customers to make intelligent decisions on what needs inclusion and what needs exclusion depending on the budget. By selecting the important requirement first, the customer obtains maximum value with the least amount spent, and this can effect a trade-off on the marginal increase in product utility with the cost to incorporate additional features. This approach also allows both the user and the customer to "pull the plug" on development at almost any time and still have highly valuable functional code, even if incomplete.
* ResilienceThe traditional approach of programming works best when requirements remain static. In actual life, requirements keep changing either because of emergence of new business opportunities or simply because the initial requirement-gathering phase was incomplete. Extreme Programming in-builds accommodation of such changed requirements through getting user stories at the start of iterations, and through feedback during the course of iterations.
* Cost SavingsExtreme Programming trims unproductive activities to reduce costs and frustration of everyone involved. It allows developers to focus on coding instead of wasting time on needless paperwork and meetings and does away with the need for separate testers.The cost of making changes increases as the software advances in its life cycle, with the cost of making changes after delivery anywhere between 5 and 100 times more than the costs of making a change at the design stage. Conventional programming methods make changes based on customer feedback at the end of the product lifecycle, whereas Extreme Programming allows changes at the development stage.
* Lesser RisksOne of the major advantages of Extreme Programming is that it reduces the risks related to programming. Conventional programming depends a lot on individual ‘superstars’ or critical members in the team. Extreme Programming, by breaking the tasks into modules, spreads the risk and reduces the dependence on any one architect, project manager, or individual coder.
* Employee SatisfactionExtreme Programming, while reducing the importance of individuals in the development process, also helps increase employee satisfaction and retention. Extreme Programming is a value-driven approach that sets fixed work time, with little scope for overtime. The breakdown of project scope into subcomponents and the constant customer feedback prevents accumulation of much work to be completed before a tight deadline.
* DisadvantagesThe advantages of Extreme Programming notwithstanding, it has its share of critics.The biggest disadvantage of Extreme Programming is that it assumes the constant involvement of the customer. Its success depends on data collection at many stages of the development process. Many customers might not be available, and many others might dislike such constant involvement.Extreme Programming code is a centered approach rather than a design-centered approach, and the lack of proper documentation creates problems in large products when project members leave and new members come in later.
* ReferencesJarvis, Bob & Gridstock, Steven. Extreme Programing (XP), Six Sigma and CMMI How they can work together. [Software Engineering Institute](http://www.sei.cmu.edu/library/assets/jarvis-gristock.pdfHutagalung), Wilfrid. Extreme Programming. [http://www.umsl.edu/~sauterv/ana...](http://www.umsl.edu/~sauterv/analysis/f06Papers/Hutagalung/Emery), Patrick. The Dangers of Extreme Programming. [http://members.cox.net/cobbler/X...](http://members.cox.net/cobbler/XPDangers.htm)

#### When to use refactoring

While refactoring is an important technique, there situations where it should not be used. The following points are relevant to this discussion:

* When adding new features
* After creating a test suite and all tests pass
* Developers plan to add more features to their software in the future
* Rule of three, Fowler (1999): it is acceptable to have one duplicate code, but when there are three or more of the same duplicate code, it is better to refactor.
* When code smells are found

#### When not to use refactoring

* Not enough time to meet deadlines. Refactoring only enhances the code, which is something that the client does not cares about. It does so little to help with programming within the short span of time. Refactor when there is time.
* Tests fail. This means the software is not working or has defects and does not comply to the agreed set of functional requirements.
* Code is too complex to refactor and it is cheaper to rewrite code. A telltale sign of this is when going through testing and finding out the software is broken and has system breaking bugs. (Fowler, 1999)

#### How to apply

The most common process for applying refactoring according to theory is Test-driven development (TDD) (Fowler, 2014). The process is done in a three step iterative cycle and the steps are: 1) Create a test for the proposed feature, it should fail. 2) Write the code for the feature in the most obvious way and run the test, it should pass. 3) Make changes to the code using refactoring techniques and without changing the behaviour of the feature to ensure the code is simple and clean.

This process is also known as red-green refactor cycle, which is shown in the figure below:

### Part B - Industry work

#### Why I chose Refactoring?

Refactoring was chosen due to my familiarity with it - I had previous experience and a pool of knowledge on the topic from undertaking BCPR301 Advanced Programming course and from the other software engineering courses, where programming principles and good practices were applicable in refactoring to achieve cleaner code. Also it fulfilled a project requirement, which was to make the code readable for other project students or developers. This is mutually beneficial to myself and future developers.

#### How I applied Refactoring

The following steps to the refactoring process were taught in BCPR301 and is based on Fowler’s (1999) steps of refactoring:

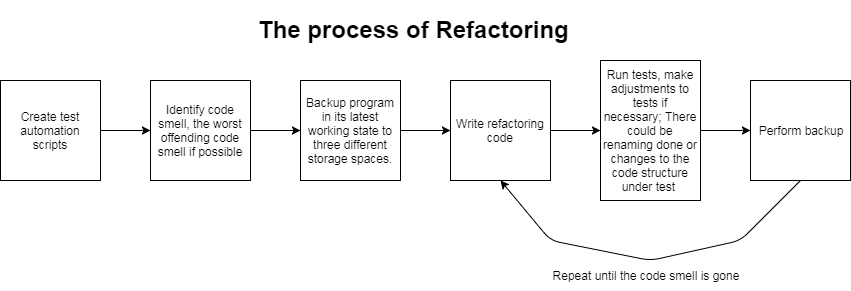


Figure 2 Refactoring cycle

Fowler (1999) says that a prerequisite to refactoring is a “*solid set of tests*”, and refactoring is to done in small steps.

#### Additional practices

When writing refactoring code, I try to go for the obvious and simple approach, as simplicity is key to readable code. I ensured that the software worked by running all my tests for the relevant code, then I used versioning (pushing latest version to the online repository or creating separate backup files of the software).

Sometimes fixing one code smell leads to creating new or discovering existing code smells, and there could be cases where a code smell cannot be removed, it is best to ignore it.

#### A specific example

This section shows a before and after example from my project:

Table 2 Before code

|  |
| --- |
| setupShowInstructionButton() {  document.getElementById("btnShowInstruction").addEventListener("click", function(event) {  this.showInstruction(event);  }.bind(this), false);  }    setupHideInstructionButton() {  document.getElementById("btnHideInstruction").addEventListener("click", function(event) {  this.hideInstruction(event);  }.bind(this), false);  }    setupBoldButton() {  document.getElementById("b").addEventListener("click", function(event) {  this.toggleBold(event);  }.bind(this), false);  }    setupItalicButton() {  document.getElementById("i").addEventListener("click", function(event) {  this.toggleItalic(event);  }.bind(this), false);  }    setupUnderlineButton() {  document.getElementById("u").addEventListener("click", function(event) {  this.toggleUnderline(event);  }.bind(this), false);  }    setupSuperButton() {  document.getElementById("sup").addEventListener("click", function(event) {  this.toggleSuper(event);  }.bind(this), false);  }    setupSubButton() {  document.getElementById("sub").addEventListener("click", function(event) {  this.toggleSub(event);  }.bind(this), false);  }    setupSubmitButton() {  document.getElementById("submit").addEventListener("click", function(event) {  this.controller.reportScoreToMoodle(event);  }.bind(this), false);  } |

##### Before

In the code snippet above, the code finds and gets the HTML DOM element by their ID value and then it attaches a click event listener and event handler to the DOM element. “.bind()” changes the scope to “this”. The “this” keyword refers to the current scope, which is View. By doing so this lets the handler call the referenced method from the View class. The code “smells” because it has a group of duplicate code that does similar things.

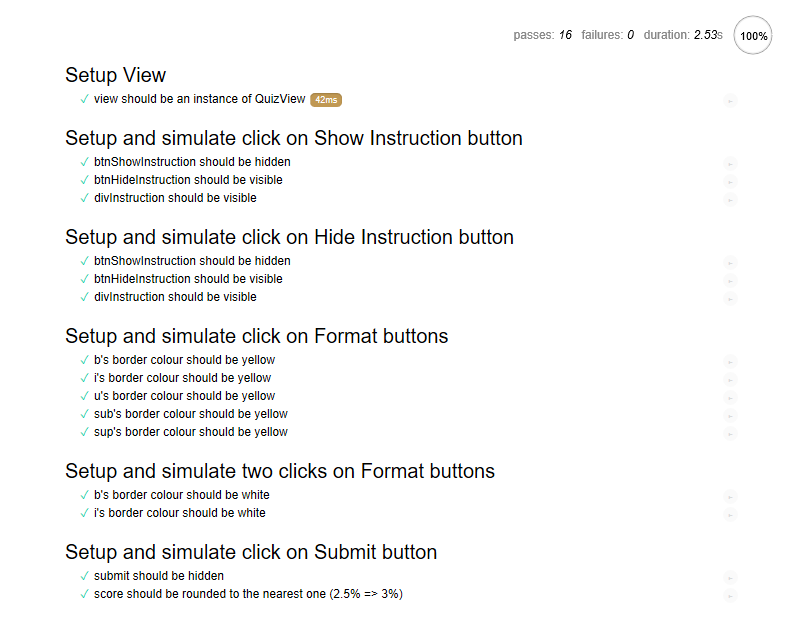
I removed the duplicate code by creating a generalized method setup in a number of steps. I created unit tests using mocha chai to test functions in the code snippet above, then I ran the tests and made sure they passed before proceeding: 

Figure 3 Tests passing

Then I backed up the program to my GitHub repository, Google drive, and made a copy of it in another folder on my physical drive, and then I started writing the refactoring code. These are the steps I followed:

Table 3 Refactoring steps

|  |
| --- |
| Step 1: I created a new method called setupControl |
| Step 2: I added a document.getElementById method to setupControl |
| Step 3: I added a parameter of controlName to setupControl, modified the line containing document.getElementById to accept the parameter value of controlName, and made it fetch the DOM element |
| Step 4: Back to the other methods in the code snippet above, I replaced document.getElementById with a method call to setupControl |
| Step 5: I compiled and ran the unit tests |
| Step 7: In setupControl, I added a variable to store the fetched DOM element |
| Step 8: I compiled and ran the unit tests |
| Step 9: Then I appended addEventListener to the newly created variable made it accept a handler as an argument. |
| Step 10: I changed the setupControl parameter list to include handler |
| Step 11: I deleted the code in the code snippet above and updated deleted method references to point to setupControl instead |
| Step 12: I compiled and ran the unit tests |
| Step 13: I changed setupControl parameter list to include subject |
| Step 14: I edited the line that contains addEventListener to include subject as an argument along with handler |
| Step 15: I added code that overrides subject to set its default value |

##### After

In the code snippet below, the setupControl method takes in three parameters, the ID value of the DOM element or controlName, a handler, and a subject. The parameter values can be passed to the called setupControl method. The handler calls the method when the click event triggers. The subject parameter has a default value of “this” and the subject can be substituted for other classes such as Controller. It specifies which class to call the method from.

Table 4 After code

|  |
| --- |
| setupControl(controlName, handler, subject) {  subject = subject || this;  let aControl = document.getElementById(controlName);  aControl.addEventListener("click", handler.bind(subject), false);  } |

Although it can be improved, the event listener is hard coded is listen to only click events and could listen for other HTML element related events such as mouse over. The getElement method could be extended to look for other element data types. Currently, it uses their ID values and could use Tag or Class names.

In summary, I followed these steps:

* Found duplicate code that could be refactored by creating a generalized function setup
* Ran tests after making changes
* Regularly performed backup

#### Refactoring in the industry

The methodology of refactoring is a recognized industry practice and is commonly applied in the IT industry whether it is done standalone or with TDD remains true to the fact (Aserg.ufmg, 2017), however the concept of code smells is not as prevalent. A survey was conducted in 2013 towards a small group of developers across several countries. Results showed that half of the respondents have heard of code smells, but some of the respondents in the group have a general understanding of code smells and they don’t apply them in their programming. (Yamashita, 2013).

For developers that apply refactoring, there are two types of tools that assist in refactoring, which are code smell detectors and automated code refactoring. Automated code refactoring is automated in a way that the code is restructured, and not that it occurs in the background as the developer is coding. It is a common practice to use tools in refactoring and doing refactoring by hand is just as common. The split in preference is based on opinions among developers. Refactoring Browser is an example of automated code refactoring tool for SmallTalk and many IDEs (Integrated development environment) such as Eclipse and NetBeans, have built-in functionalities or installable plugins that provides automated code refactoring. The tools cover common refactoring techniques such as Rename Method, Pull Up Method, Extract Method, etc.

JSNose is an example of code smell detection tool.

### Part C - Compare and contrast

In this section, I will compare and contrast my process of refactoring with theory and recognized industry practice. To recap, in this project, I chose to identify the code smells manually, refactor the code manually, and applied a variation of Fowler’s refactoring steps.

#### Fowler

According to Fowler (1999), he puts emphasis on taking small steps: “*test, small change, test, small change, test, small change*”, which I put into practice in my code example. After making small incremental changes in my code to the point where my program is working, I ran tests for defect prevention purposes. Testing ensures that the part of code being tested meets the defined functionalities and no defects have been created during the refactoring process, and also testing checks that there are no visible changes to the behaviour of the code, but only changes to the internal structure. Any changes to the behaviour should introduce a defect and the test process will fail. An advantage this has over TDD is that tests don’t have to be maintained during refactoring with the exception of updating test to reflect changes to code structure and renaming. It does not matter how poorly written the tests are, as long it captures the behaviour of the code.

A benefit that TDD has over my refactoring process is that the code coverage should be near 100%. The code is covered by tests as it is developed. In order for my refactoring process to reach a high percentage of code coverage, it has to rely on fulfilling the four criteria when designing unit tests, which are:

* Function
* Statement
* Branch
* Condition

#### Code smells

Using Fowler’s list of code smells. I’ve discovered four different code smells in the existing JavaScript code, which were Long method, Temporary field, Duplicate code, and Dead code.

The most common code smell was Duplicate code. This shows that while Fowler identifies a long list of smells, you do not always find all of them.

#### Tools

I didn’t use code smell detection tools because I wasn’t aware of its existence and other developers say these tools are unintuitive and untrustworthy. Also I did not use any automated code refactoring tools, as Notepad++ did not have a relevant plugin. I could have used a code smell detection tool to speed up the task of identifying and locating code smells due to its instantaneous detection time. It could be useful in detecting code smells that developers are oblivious to when they are focused on the implementation side of coding. Although, it does have its limitations. According to researchers (Fontana, et al. 2012), they found that code smell detection tools are not reliable and may give mixed results. The reasoning behind this that they gave was that the definition of code smells is deemed ambiguous such as “many”, “long”, etc. There is no defined metric of how many lines of code is considered too much. Developers have their own interpretations and possibly have created the tools in this regard. For example, to some developers, a class that has least eight methods may be considered a code smell or it could be ten fields to other developers. Another reasoning is that different tools could have different detection methods. The methods used could be based on defined metrics that are specific to the code smell or some complex algorithm to calculate the metrics.

#### Usefulness

I found the methodology of refactoring to be not so useful because I benefited little from it and I lost development time. For example, the “before” code had an instruction button that was named “inshn”, a method was named “coun”, and many more that were similarly named, which I found they weren’t memorable, meaningful, and I had to refer back to the HTML file to find out how to spell it. After refactoring using the Rename method, “inshn” became “instruction” and “coun” became “reportScoreToMoodle”. I did find it easier to identify and type the name of DOM elements after refactoring, but it didn’t save me a lot of time in return. Refactoring should be done when time constraint isn’t detrimental to the project’s success. The impact of renaming may seem miniscule at the moment, but it will act as a catalyst for maintenance of the software in the future. The goal of refactoring is to make the system less prone to errors and to reduce the number of defects that developers make when programming, which will enhance productivity. Future developers will appreciate the effort and steps taken to make it easier for them to program.

### Conclusion

Refactoring is a great practice to have in most circumstances. When time isn’t a primary concern and when backed up by tests, there is little risk in applying refactoring. Overall, refactoring comes with benefits that outweigh the limitations. Code smells should be considered when applying refactoring. In hindsight, I would continue to use refactoring if I had the opportunity to complete the same project again. Although, the benefits for me were not realised in the project, I learned that refactoring will improve the useful life of the software and give developers a better programming experience.

### 

# References

A, Y., & Moonen, L. (2013). *Do Developers Care about Code Smells? An Exploratory Survey*. Retrieved from simula: https://www.simula.no/file/codesmellsurveypdf/download

aserg.ufmg. (2017). *What are the most common refactoring operations performed by GitHub developers?* Retrieved from aserg.ufmg: https://medium.com/@aserg.ufmg/what-are-the-most-common-refactorings-performed-by-github-developers-896b0db96d9d

Brodie, L. (1984). *Thinking forth.* USA: Punchy Publishing.

Fontana, F., Braione, P., & Zanoni, M. (2012). *Automatic detection of bad smells in code: An experimental assessment. In Journal of Object Technology*. Retrieved from Journal of Object Technology: http://www.jot.fm/issues/issue\_2012\_08/article5.pdf

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code.* USA: Addison-Wesley Professional.

Fowler, M. (2014). *Workflows of Refactoring*. Retrieved from Martin Fowler: https://martinfowler.com/articles/workflowsOfRefactoring/