HOW CLEAN IS OPEN SOURCE CODE?

VIDMANTAS BLAZEVICIUS

This dissertation was submitted in part fulfilment of requirements for the degree of MSc Information and Library Studies

DEPT. OF COMPUTER AND INFORMATION SCIENCES

UNIVERSITY OF STRATHCLYDE

SEPTEMBER 2015

DECLARATION

This dissertation is submitted in part fulfilment of the requirements for the degree of MSc of the University of Strathclyde.

I declare that this dissertation embodies the results of my own work and that it has been composed by myself. Following normal academic conventions, I have made due acknowledgement to the work of others.

I declare that I have sought, and received, ethics approval via the Departmental Ethics Committee as appropriate to my research.

I give permission to the University of Strathclyde, Department of Computer and Information Sciences, to provide copies of the dissertation, at cost, to those who may in the future request a copy of the dissertation for private study or research.

I give permission to the University of Strathclyde, Department of Computer and Information Sciences, to place a copy of the dissertation in a publicly available archive.

(please tick) Yes [✓] No [ ]

I declare that the word count for this dissertation (excluding title page, declaration, abstract, acknowledgements, table of contents, list of illustrations, references and appendices is 18894.

I confirm that I wish this to be assessed as a Type 1 2 3 4 5

Dissertation (please circle)

Signature: Vidmantas Blazevicius

Date: August 25th, 2015

ABSTRACT

Context: In the last decade, the rapid shift towards agile techniques has greatly impacted the industry of software development. Developers started to get pressured for more and more quality source code in less amount of time. Many software experts have come up with the principles and practices to be followed in order to make sure that source code is more understandable and prone to enhancements. But are these practices and principles actually followed in the real world projects, and if they are, to what extent?

Objective: The goal of this study was to identify a set of lower level characteristics and principles that make source code more readable and understandable and determine to what extent they are followed in selected open source systems.

Method: Custom analysis tool was developed in order to measure certain code metrics, such as average variable, class or function name, the amount of polyadic functions within systems, average amount of statements per function. Using the combination of this custom analysis tool and a 3rd party tool called SourceMonitor, a set of 20 open source systems were selected based on their size and analyzed.

Results: Commenting and especially the public API documentation remains to be overused by majority of the systems. Software size seems to be affecting quite a few other metrics, such as the average number of function parameters and average complexity. Shorter functions with few parameters are preferred across the board by all systems. Short name variables are nearly extinct and are barely used.

Conclusions: With the exception of commenting, most guidelines and principles have indeed been followed by the selected open source systems to a decent degree. Some systems perform better in regards to certain metrics, signifying difference in emphasis on particular principles and techniques by developers. Future work areas identified include analyzing different aspects of code cleanliness, analyzing the semantics of comments in order to determine their usefulness as well as semantics of function names to discover whether the function implementation is appropriate for the given name, determining the value of each 'Clean Code' characteristic to the overall cleanliness of the code and studying incremental growth of large software systems in order to find out whether the guidelines and principles are followed throughout the development lifecycle and to what extent.

TABLE OF CONTENTS

TABLE OF FIGURES

CHAPTER 1: INTRODUCTION

Object oriented programming has become incredibly dominant in the industry of software development (Cass, 2015). Along with it, principles and practices for writing most efficient and understandable code were derived from the experience gathered by developers around the globe. Quite a few software experts have composed their knowledge into textbooks that are supposed to act as handbook for programmers guiding them through the process of learning and improving important software development skills such as Refactoring: Improving Design of the Existing Code (Fowler, et al., 199), Code Complete 2nd Edition (McConnell, 2004), Agile Principles, Patterns and Practices in C# (Martin, 2002), Professional Refactoring in C# & ASP.NET (Arsenovski, 2009). Also there exists countless online blogs and articles where developers share the most important techniques and principles and explain the rationale behind them. On top of that, new principles and techniques are being proposed by research community. However, there seems to be a lack of empirical studies analyzing exactly how much each of those techniques and principles have been adopted and to what degree.

This study consists of reviewing lower level principles, primarily the ones listed in the textbook Clean Code: A Handbook of Agile Software Craftmanship (Martin, 2009), and an empirical analysis using a combination of custom and 3rd party tool in order to find out the extent of adoption of those principles in open source software projects. The study focusses on the source code properties that affect its readability, understandability, testability and reusability.

The aim is to find out the extent of usage of certain practices and principles that make software easier to read, understand, test and reuse is motivated by a number of goals:

- It is of potential interest to software experts to discover the degree of actual usefulness of their suggested practices.
- It is of particular interest to software project managers to find out the indicators of potential violations of practices based on the code metrics, especially considering the direct relation between the quality of the code and the overall cost of developing and maintaining software.
- It is important for software developers to understand likely reasons of violating certain practices and to learn from trending mistakes.
- It is important for research community to know which practices are adopted the most and to what extent in order to help further debates and propositions.

This study is further motivated by the goal of contributing to the statement in the foreword to a textbook Clean Code: A Handbook of Agile Software Craftmanship (Martin, 2009):

> "Attentiveness to detail is an even more critical foundation of professionalism than is any grand vision. First, it is through practice in the small that professionals gain proficiency and trust for practice in the large."

Empirical analysis of certain code metrics helps to realize exactly how attentive to detail nowadays developers are. Also, useful insights and observations can then be drawn on the relations between particular details.

Moreover, the shift towards agile techniques within the industry of software development has caused additional pressure on developers when writing the code which results in potential early abandonment of the code (Heusser, 2013). Agile techniques are all about timelines, meeting deadlines and organizing

tasks efficiently. One could say it is about moving forward with the project as fast as possible. However, the unexpected consequence is that there is little importance placed on looking back and reflecting on what has been done already. Coupled with issue of early abandonment of the code mentioned earlier, this opens an interesting area to study. Making existing code more readable and understandable becomes significantly more important when the timeframe for developing new code gets smaller.

To address these goals, key advices (primarily the ones given by Robert C. Martin) on how to follow principles of writing clean code were identified. Most important characteristics that make code more readable, understandable and modular were reviewed. These characteristics were mainly low level ones, such as naming, functions, comments, etc. A few examples with the combination of custom source code as well as real world project source code were selected and adjustments were made in an attempt to apply described practices.

Custom analysis tool was developed using JDT library in order to measure certain code metrics. Twenty open source systems were then selected based on their size and using the combination of custom and 3rd party analysis tools 17 code metrics were measured. The resulting data was used in order to discover interesting relations between particular metrics as well as determine the extent to which the selected open source systems have adopted principles and practices that are suggested by software experts and aim to greatly increase overall source code readability and understandability.

Such empirical analysis showcased that the selected open source systems have indeed been following the characteristics identified to be ones of the most important when it comes to writing clean code. The only exception was comments in the source code – even though quite a few guidelines to commenting suggest minimalistic approach, it seemed that the open source systems ignored this by cluttering their codebase with useless comments. Besides that, the results show that: short name variables are nearly extinct and used only very rarely; it is possible to produce large codebases while following clean code practices to a decent degree; software size seems to be affecting quite a few other metrics, such as average number of function parameters and average complexity; developers seem to prefer shorter functions with few parameters which greatly impacts testability.

In the following section, the ‘Clean Code’ term is described in more detail, identifying common general characteristics that different software experts claim ‘Clean Code’ to be. Each of those characteristics are then also explained in regards to their importance. Then follows the review of the lower level principles that contribute to ‘Clean Code’ along with the related work section where existing material and research on ‘Clean Code’ and the mining of software repositories is summarized. The analysis tool section outlines how certain source code metrics were measured programmatically, including the libraries used and general techniques that are used for automatic source code inspection. Afterwards, the analysis results section contains the list of selected open source systems and discusses the data gathered after they were analyzed with observations and insights where possible. Study concludes with an argument on threats to validity, a succinct of the analysis results and discussion on future research.

1.1 Clean Code

“Software development is a design process. A fundamental property of designs is that at the start of the design process, the final outcome is uncertain” (Baldwin & Clark, 2005). Such uncertainty is very problematic when developing open source software nowadays. It is obvious that the nature of open source projects is much different to the closed source ones. When it comes to open source projects, the

code usually involves contributions from many developers which come from different backgrounds and are not really enforced to follow certain development techniques. Therefore, the resulting project can be considered as a mashup of different styles of coding, different preferences when it comes to choosing variable or method names. Even though there exist conventions in the world of software development, the nature of open source projects imply that these conventions will most likely be ignored.

According to Baldwin and Clark, "the architecture of a codebase is a critical factor that lies at the heart of the open source development process" (Baldwin & Clark, 2005). While this is a valid point of view and there has been a lot of research conducted as to what the correct architecture should be, it suggests that the grand vision is the most important thing. Different software architects will probably have different visions in describing what it means exactly, but in reality, the overall architecture is made of small things and it could be implied, that these small things matter just as much, therefore detail and attentiveness to detail is a critical foundation of a good architecture.

Coplien and Bjornvig gives an example of German architect Ludwig Mies van der Rohe who once said "*God is in the details"* (Coplien & Bjornvig, 2010). Rohe is known for this quote because he was very careful and attentive to smallest underlying things behind the architecture of his buildings. He would personally select every doorknob and every bathroom tile for the buildings he designed because Rohe believed that small things matter. Unfortunately, nowadays this is not really a main concern in programming as the main focus usually lies on delivering a product with an appealing front face rather than what the inside looks like. This is even more of a problem for open source software as the developers are less concerned about the quality of their code and how it merges with the rest of the system.

Clean Code is a term used in this study and it refers to Robert Martin's book – "Clean Code: A Handbook of Agile Software Craftmanship" (Martin, 2009). He doesn't give a single definition of what *clean code* actually is, because he probably can't. The term is just way too generic for a single definition to exist, so instead, he gives a variety of definitions from different well known software experts. There are, however, similarities in what those experts claim *clean code* to be and the main general characteristics identified between various definitions are as follows:

- Easy to understand.
- Easy to modify.
- Easy to test.
- Contains no duplication.
- Expresses all the design ideas that are in the system.
- Code that is reusable.

1.1.1 Easy to understand

Source code can be considered as a form of communication, and even though it is written so that computers can interpret it and perform certain functions accordingly, it is just as important for people that work on the same project or may contribute to the project later in the future. Thus in order for those people to understand the code easily, they should able to read it with little amount of effort.

A lot of factors contribute to making the code easily understandable. Robert C. Martin seems to emphasize more on the details of the actual source code and how it is written, saying that every brace placement, every indentation and even every variable name that a programmer choose to use should be carefully thought about because it is eventually going to make up a larger project and if all the small details are in

their appropriate places then the overall picture will become much more easier to understand. An interesting study was also conducted by Vineet Sinha's team and it aimed to find out exactly what makes codebase easy to understand (Sinha, 2011). They have organized a survey with the help of developers that have worked with large codebases. The findings of this survey showed that easy to understand code is usually well documented, contains examples and articles describing of how to use the code. On top of that majority of the developers that participated in the survey said that the projects they worked on were "lacking the high level overviews that would allow them to better understand a new project or codebase at a glance" (Sinha, 2011).

The difference on emphasis as to what an easy to understand code should be is clear. Probably no one would argue that both, small details and high level overview diagrams and examples, are equally important contributors thus the combination of these two factors can be considered the characteristics that make code easy to understand.

1.1.2 Easy to modify

Nearly every developer that has ever had to add additional features to a program must have come across a situation where he wished that the code he is attempting to modify and enhance would have been designed to be extensible from the start. David A. Thomas, founder of *Object Technology International* includes "easy to modify" gave his definition of what "Clean Code" is. He says that "c*lean code can be read, and enhanced by a developer other than its original author*" (Martin, 2009).

It is fairly hard to put properties on what makes code easy to modify. Usually, code that is written for general purpose is much easier to reuse and modify. Easily extensible code should also inherit all the characteristics of easy to read and understand code, because one must first understand what the code does before he even attempts to enhance it with additional functionalities, therefore these two "Clean Code" properties are tied together.

1.1.3 Easy to test

Ron Jeffries, author of "*Extreme Programming Installed"* and "*Extreme Programming Adventures in C#"* is a big activist when it comes to testable code. He ties "Clean Code" to tests and goes to say as far as that the code without tests is a bad code regardless of how readable or understandable it is. It is no surprise since Test Driven Development has been a mainstream discipline in the industry of software development. Code coverage is an important metric when it comes to testing and it shows the portion of the source code has been actually tested. Ideally, following the main rule of Test Driven Development discipline, the entirety of the codebase should be covered since the rule says that you cannot write production code before you have written unit tests for that code. What is often overlooked, however, is how to actually write a code that is easily testable.

Throughout the years many developers have created different methodologies of how to write testable code. There exist many books and articles online sharing the best practices and disciplines that a developer has to supposedly follow in order to be successful at writing testable code. It would be impossible to select one methodology that is best because all of them come from different experiences, different people that have worked on different problem domains. Below is a list of practices that have been mentioned in nearly every methodology available and should give an overview of what easy to test code should be like:

- Separation of concerns – ensuring that different parts of an application are appropriately separated contributes highly to writing code that is easily testable. Each of those different parts should have API's to allow interaction. Usually the application can be separated into different areas like displaying the data, handling the events, retrieving and processing the data. These areas can then also be broken down further into smaller modules. Such separation of concerns helps a lot in maintaining code that is highly testable by making it possible to create mock objects in order to replace different components of the application and test isolated units of code.
- Object-oriented code design allows using constructors that can reproduce objects on demand ensuring minimal state transmission between different tests.
- Loose coupling / dependency injection is another practice highly emphasized in methodologies for writing testable code. It is fairly obvious that components that are not explicitly dependent on other modules are much easier to test since it is then possible to use dependency injection and pass compatible mock objects into components being tested.
- Elimination of global state increases the ability to test components in complete isolation since the main point of unit testing is to test different parts of application separately.

1.1.4 Contains no duplication

Probably any developer would say that duplication is one of the main factors when it comes to discussing "Clean Code". Code duplication causes the amount of source code lines to become larger than it needs to be, and the size of the software generally has a high influence towards the maintenance cost and effort. It is not uncommon for developers to produce duplicated code, especially when under pressure. In fact, it is a favored technique and sometimes even considered as reusing the code, because most of the time code duplication is not an exact copy of some chunk of code, but an adaptation of already written and tested code that achieves the required functionality criteria.

Because code duplication sometimes might seem so innocent since it deceivingly achieves greater short term development speed, it is often an overlooked factor and will eventually result in significantly larger software size and higher maintenance cost. However, there are hardly any practices or principles to follow when writing the code to avoid code duplication. Everything is fairly circumstantial and code duplication may sometimes be a necessary evil. There exist quite a few tools available that can detect potential code duplication and at the moment the use of such tools seems to be preferred way to combat code duplication in the industry of software development.

1.1.5 Expresses all the design ideas that are in the system

Such a vague statement, yet at the same time, so honest, indisputable and justifiable, that it would be hard to not include it in "Clean Code" description. Ward Cunningham, inventor of Fit, inventor of WiKi and coinventor of eXtreme Programming said that "*you know when you are working on a clean code when each routine you read turns out to be pretty much what you expect"* (Martin, 2009). Ward pretty much hit the jackpot with this quote since if all the source code would accurately express design ideas in the system, it would inherently be much easier to understand, modify and test.

However, it is really hard to achieve such cleanliness in the source code since this characteristic of "Clean Code" is not about having good design, but actually expressing every design idea incredibly well during the development stage. It is a combination of all small things: class names, method names, their implementations, and other "Clean Code" characteristics that have to be carefully and appropriately

thought of before a reader can actually say that a piece of chunk code he read was exactly what he expected and if he tried to improve on it, he would have no success and result with the same solution.

1.1.6 Code that is reusable

When object-oriented programming languages became mainstream, the industry of software development has become fully equipped with tools to produce reusable code. Developers that have mastered the techniques of writing reusable code find themselves actually doing much less work because they just applied one generic solution to different problems. A common misconception among developers is that reusable code is considered to only be put in libraries or middleware. In fact, it comes in many different forms. Even when using operating system function calls or a piece of code that our coworker has written for a different project should be considered reusing the code because that's exactly what's happening.

Producing high quality reusable code requires experience because you have to make sure that it solves a generic problem correctly while meeting multiple needs. There exist countless practices and guidelines available online where experienced developers share their knowledge gathered throughout the years on how to write reusable code. All of them, however, boil down to following single responsibility principle which states that "every class should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class" (Wikipedia, 2015). While entire classes can be written to be reusable, it is usually only several functions that are intended to be fully reused. But the single responsibility principle holds for such functions as well, and suggests that each function should do one thing, but do it very well. Code that is written in such manner is asking to be reused. Any problem that is tackled by specific function is bound to occur again at some point. That is where such code becomes incredibly valuable, because it can be reused.

CHAPTER 2: RELATED WORK

There has been a fairly limited amount of previous academic research in this field, especially empirical studies. This section is separated into two major themes: 'Clean Code' and mining software repositories.

2.1 'Clean Code'

In the definition of terms section, a set of main characteristics of what 'Clean Code' is has been identified based on similarities from definitions by various software experts. The understanding has changed greatly over the last decade.

Bjarne Stroustrup, inventor of C++ and author of many publications regarding development within C++, has defined his version of 'Clean Code' as a logic that is straightforward and makes it hard for bugs to hide (Stroustrup, 2008). He also emphasizes on having minimal amount of dependencies in order to ease the maintenance as well as optimize performance. He closes his definition with the assertion that clean code should do one thing, but do it well. It is interesting because a lot of principles within the overall software design can be applied to this simple statement. The ability to design the system, each of its functions, classes and modules with a focused attitude has, however, decayed from the scenery of software development. The principles have shifted to the attempts to write a code that is modular, can perform multiple tasks, which in turn makes it more ambiguous in regards to its purpose. With such change Stroustrup started discussing how to achieve both, modularity and cleanliness of code, while maintaining the elegancy of single-mindedness and reducing the level of ambiguity. He mentions that the key to keeping the code clean is caring about the code: returning to the already written code and fixing it regularly according to the newest standards, carefully rethinking error handling and emphasizing on merging the new code with the old code seamlessly (Stroustrup, et al., 2010).

Ostwold and Host published a paper, where they investigate existing naming conventions and propose a mechanical check to determine whether the method's name and implementation is a good match (Host & Ostvold, n.d.). They use implicit conventions within the large amounts of software written in Java for matching method names and implementations in order to extract rules for naming methods. The methodology in this publication consists of decomposing the software into methods and using the names of those methods in a tagging process, where each fragment of a method name is assigned with a predefined tag based on what it means. Such decomposed, tagged method names are then analyzed in regards to the relevance of the method semantics. Based on the analysis, a set of phrase-specific implementation rules is derived. The automatic suggestions becomes trivial as the method semantics should then match the derived implementation rules and if they don't, the chosen method name is not appropriate for what function the method is attempting to do. While there exist problems in this approach, it is certainly an interesting way of validating naming of methods within the code. The main issue comes when trying to tailor such automated process to different programming languages. Even though the naming conventions persist through different languages, there are several occasions where the use of specific language can make the name of the method may look irrelevant to its semantics, but in fact, in the bigger picture and taking into account the dependencies, the name would be sound.

Weimer and Buse "explore the concept of code readability and investigate its relation to software quality" (Buse & Weimer, 2010). They have conducted a study where they had over 100 participants from The University of Virginia trying to make judgements on whether the code is readable and understandable or not. Interesting outcome of this study showed that people tend to settle on the same opinion about what

readable and understandable code is. Weimer and Buse then attempted to establish a model with a set of features that a readable code should have. They have considered metrics such as: line length, number of identifier, identifier length, indentation, comments, etc. This model was then applied to a number of SourceForge projects to test the correlation between assumed readability and the actual quality of the software. The results showed clearly that the code which had high level of readability was less likely to contain bugs and errors.

Collar and Ricardo explore the correlation between readable software and the cost of development (Collar & Valerdi Ricardo, 2012). Their study shows that the extent of which readable software affects the cost of development increases over time and is especially significant at the last stages of project life cycle. These results are especially important to larger projects where multiple developers can contribute to the development of the same modules or there exist a changeover of development staff. To support the study, they have experimented with participants that had varying knowledge of software development and presented them with the code that was different in its semantic implementation but had the same purpose. Participants were asked to extend the code with an additional function, which required them to firstly understand the existing code. The experiment showed that independent of programming domain knowledge, readable code was much easier to understand and build upon.

Sridhara G. and her colleagues proposed an approach to automatically detect and describe abstraction levels aiming to increase the developer's ability to comprehend the source code (Sridhara, et al., 2011). They devised a heuristic method to identify sequences of code statements that have commonalities in terms of actions performed. Usually commonalities are signified by similar function calls. Such blocks of code are considered to stay at same abstraction level. Moreover, their approach goes further and attempts to find a general description that would characterize a given block of source code. An evaluation of this approach was done on numerous open source Java systems with a group of independent evaluators to judge the precision of the prototype. Results were very promising and the prototype was able to accurately identify blocks of codes with commonalities for over 75% of the cases. While the current version of the algorithm only works at a function level, it seems that it could be improved and adopted to work at class, package or even system level. Also, nature of this research suggests that it could be a useful functionality for IDE's to help developers during the design stage.

McBurney and McMillan proposed an interesting technique to automatically generate documentation of function context for source code (McBurney & McMillan, 2014). They identified that previous attempts were lacking the ability to explain the context of source code and. Proposed approach consists of extraction of keywords about the statements within function's body and the usage of a custom algorithm to generate English descriptions. They have evaluated this approach with the help of some students as well as professional programmers in order to determine the degree of accuracy of the generated documentation. While the results have shown a clear superiority to comparable approaches, the qualitative analysis has revealed a potential drawback of producing too much unnecessary information. With that said, a tool with a refined version of such technique could certainly find areas for applicability by lessening the burden to write documentation comments from developers.

Kuhn, Ducasse and Girba introduced a technique called *Semantic Clustering* which is based on "Latent Semantic Indexing and clustering to group source artifacts that use similar vocabulary" (Kuhn, et al., 2007). This technique is aimed to increase understandability of software systems by deriving common topics from identifier names and comments which in turn reduces the amount of source code developer has to

read before becoming familiar with a given system. Some case studies of the systems that contained legacy with old naming conventions presented a challenge to preprocess data, but overall results revealed that this approach is great at identifying domain specific languages, their distribution and usage in the system.

2.2 Mining software repositories

In order to perform empirical studies, the data from the software repositories needs to be extracted. It is currently available for most large open source projects and it represents a record of development of those systems. Unfortunately, software repositories are not designed to facilitate empirical understanding of a software project due to the software tools containing various anomalies and issues in their recorded information (Hassan, et al., 2010). Hassan et al. in discuss the approach of transforming software repositories into a more active model which can then be used to empirically validate theories and models of software development using the information in these active repositories (Hassan, et al., 2010). They give an example of Amazon.com shopping website, where Amazon is using customer's history of purchases in order to make smart suggestions. The active repository model would employ similar technique to source control change history in order to produce recommendations for developers.

Another approach is the use of statistical topic models on the data gathered from software repositories. Stephen Thomas explores such use of statistical topic models in an attempt to automatically discover structure of textual repositories (Thomas, 2001). He bases the need of his research on the ever growing complexity of the overall source code that comes with the evolution of a software project.

CHAPTER 3: 'CLEAN CODE' CHARACTERISTICS

This section will review lower level principles, guidelines and ideas as to how 'Clean Code' can be achieved and will heavily reference Robert C. Martins book "Clean Code: A handbook to agile software craftmanship" (Martin, 2009). Even though the book is specific to Java language, and some 'Clean Code' characteristics would look slightly different in other languages, most guidelines can be applied generally anywhere in software development. Moreover, Robert C. Martin pointed out himself, that this book contains knowledge gathered throughout many years of his software development career and is not necessarily absolute correct way to write 'Clean Code', therefore it is entirely likely that there exist alternative practices would actually disagree with what the book is suggesting.

3.1 Names

Software development would be a very hard process if we didn't have some clever way to call things, some convention that developers would agree to and follow. Names are absolutely everywhere: every function, every variable and every class in any software project should be chosen for a reason. The amount of relevance that any name holds to what it is actually naming can highly increase code readability and therefore names are one of the most important low level 'Clean Code' Characteristics.

Robert C. Martin has singled out the following characteristics about names: intention-revealing names, avoiding disinformation, meaningful distinctions, pronounceable names, searchable names, avoiding encodings, avoiding mental mapping, class names, method names, one word per concept, using solution domain names, using problem domain names, meaningful context, no gratuitous context (Martin, 2009).

3.1.1 General principles

Classes, functions and variables should have names that reveal their intention by providing the context of what are they going to be used for (for example, set of variables with names *a, b, c* wouldn't reveal anything for the code reader while names *year, month, day* would instantly suggest that the variables will be used in describing some date). Classes and objects should have noun or noun phrase names while functions should have verb or verb phrase names. Moreover, names should be used within consistent spelling, because inconsistent spelling creates disinformation. Programmers should avoid using characters like lower-case *L* or upper-case *O* as variable names, especially in combination since they look exactly like the constants one and zero. On top of that, using noise words and adding number series just to make the source code compile should be avoided at all costs. Different names should hold different meanings. A lot of examples can be found in functions that copy some data between data structures. It is very common for developers to choose parameter names for those functions as *a* and *b* or *a1* and *a2* because the functions are generally fairly simple and the intent is already revealed within function name but simply changing those parameter names to *source* and *destination* adds so much to readability, especially in the cases where there is a specific algorithm within the function body. Apart from agreed conventions, noise words such as *info, data, object, a, the,* are redundant in most cases and very heavily overused by programmers nowadays.

Noise words, number series and in general, single letter or very short variable names creates the requirement for the reader to perform heavy mental mapping. It is usually seen in the parts of code where there is some algorithm implementation. In order to understand what such names mean reader has to mentally map the variable name to what it actually means and each time they see that variable they will have to think twice before realizing what it does. To make matters worse, if the scope of such poorly

named variable is relatively large, a lot of unnecessary mental work can be created for absolutely no reason. Code snippet 3.1 is taken from graphics framework JHotDraw (Anon., n.d.):

```
Rectangle r1 = start().owner().displayBox();
Rectangle r2 = end().owner().displayBox();

int dir = Geom.direction(r1.x + r1.width/2, r1.y + r1.height/2,
          r2.x + r2.width/2, r2.y + r2.height/2);
```

Code Snippet 3.1. JHotDraw - ElbowConnection.java

It wouldn't take much effort for anyone to realize that this piece of code is used to determine relative direction between two rectangles. The original code has those two rectangles named as *r1* and *r2*. Neither of these names are distinctive enough to provide any sort of information of the developer's intention for these variables. Of course, a reader will quickly realize that there is a connection between these rectangles, and *r1* corresponds to the rectangle at the start of this connection while *r2* corresponds to the rectangle at the end. There is, however, absolutely no reason not to refactor these two variable names into something that is meaningful and readable. After renaming those variables to *rStart* and *rEnd* the resulting code is as follows:

```
Rectangle rStart = start().owner().displayBox();
Rectangle rEnd = end().owner().displayBox();

int direction = Geom.direction(rStart.x + rStart.width/2, rStart.y + rStart.height/2,
                rEnd.x + rEnd.width/2, rEnd.y + rEnd.height/2);
```

Code Snippet 3.2 JHotDraw – ElbowConnection.java (modified)

Even though its only 3 lines of code, it is so much easier to read when you don't have to mentally associate the variable name in the code with its actual meaning. Furthermore, this example can still be built upon by adding another abstraction level. It is fairly difficult to understand how the *Geom.direction* function is used exactly. Even though it is obvious that it takes four coordinates as parameters and returns an integer flag that corresponds to certain direction (north, west, east, south), it would be much easier to either have additional overload of *direction* function that takes two rectangles as parameters or encapsulate the parameters passed in well-chosen names.

```
Rectangle rStart = start().owner().displayBox();
Rectangle rEnd = end().owner().displayBox();

int direction = Geom.direction(getRectangleCenterX(rStart), getRectangleCenterY(rStart),
           getRectangleCenterX(rEnd), getRectangleCenterY(rEnd));

private int getRectangleCenterX(Rectangle rectangle){
   return rectangle.x + rectangle.width/2;
}

private int getRectangleCenterY(Rectangle rectangle){
   return rectangle.y + rectangle.height/2;
}
```

Code Snippet 3.3 JHotDraw – ElbowConnection.java (modified)

Two extra functions were added for the sake of clarifying the intention of *Geom.direction* function usage. Even though this might seem like an unnecessary tweak, it actually completely removes any need for the

reader to try and figure out what the code does. It is now obvious that the variable *direction* holds the meaning of direction between the center points of two rectangles. In fact, the codebase was also just equipped with the reusable piece of code that calculates the center coordinates of a rectangle that can be moved to utility classes if needed.

There is a common pattern within these lower level principles and guidelines of choosing names – short names are simply not expressive enough. Single letter names or the usage of numeric constants for variable names creates problems when performing reference search, especially if such variables are used for large scopes. Meaningful names should always be preferred even though single letter names could technically be used as local variables inside shorter functions without much affect to readability. It is worth mentioning, however, that there is a danger of inadvertently adding redundant context to a name. Using pattern, algorithm names, computer science or math terms as well as names drawn from the problem domain is highly encouraged considering that the source code will be read by other programmers or coworkers that are already familiar with the same problem domain. The general rule of thumb is to keep names as short as possible without damaging their ability to be clear and understandable to the reader.

3.1.2 Pronounceability, encodings and concept naming

Another obvious naming principle, but very often overlooked, is the usage of pronounceable names. Legacy software codebases triumph in providing examples of bad naming examples. It is very likely to come across names like *Lvlcmprto, TCPGen,* etc. which actually are so hard to read and realize what they mean (in this case *TCPGen* means *Generic Test Case Suite Prioritization*). Just renaming such variables into a pronounceable form, like *levelCompareTo* and *genericPrioritization* increases readability by a lot and avoids wasting reader's time to find out what the variable actually means.

Encoding type or scope information into names should be avoided according to Robert C. Martin since it simply adds the extra burden of deciphering those names and therefore it is a counterproductive effect to later ask each new employee to learn these encoding languages. This particular principle is not that relevant anymore since such encodings are barely used anywhere nowadays and the main source where you can find examples is once again – legacy software.

Defining concepts can be fairly tricky. One concept can be expressed through several different words sometimes. A good example is using *fetch, retrieve*, *get, set, load* predicates to functions from different classes. It would confuse the reader since it is hard to remember which one belongs to which class. Even though IDE's nowadays provide context-sensitive clues such as what methods belong to which class, but it is still a better practice to stick with one word per concept.

```
Bad Code
void LoadSingleItem();
void FetchItemsFiltered();
void GetAllItems();

void SetItemsToView();
void SetItemValue(int value);
```

```
Clean Code
void GetSingleItem();
void GetItemsFiltered();
void GetAllItems();

void LoadItemToView();
void SetItemValue(int value);
```

Code Snippet 3.4. Custom example of concept naming

3.2 Functions

Functions are one of the most important parts in any kind of software project. They hold the implementation of design ideas and therefore expressing the intention of functions is incredibly important. It is really hard to understand and modify the source code if the functions are not organized properly. General 'Clean Code' characteristics for functions are size, purpose, argument count and duplication.

3.2.1 Size

In general, functions should be small. However, there is no hard rule that determines the absolute best amount of lines per function. Robert C. Martin specifies that functions should hardly ever be 20 lines long. Function size directly affects readability. The longer the function is, the harder it is going to be for the reader to understand the intention. In fact, longer functions will most likely contain several abstraction levels and will break single responsibility principle, which states that states the function should do one thing and only that thing. Furthermore, long functions is like a horror scenario for developers when trying to find and fix bugs. With that in mind, there are scenarios when it is hard to get away without writing long functions. The main one is when using switch statements – the nature of switch statement will most likely increase the size of the function, but might not damage readability.

3.2.2 Purpose

Single responsibility principle is mainstream in software development nowadays and highly emphasized by most developers. Ensuring that the function does one thing is not easy. Function should have a descriptive name that reveals that one thing and then the body of the function should appropriately do that one thing without derailing to other tasks. The biggest sign for a function doing more than one thing is when it contains *and* or *or* in its name, then it is most likely not going to do one thing. On top of that, ensuring that the statements within the function are at the same abstraction level will also help to achieve a function with sole purpose. Often we see a mix of abstraction levels within functions that do many things and they are very confusing to read. Another sign of breaking the single responsibility principle is being able to reasonably divide functions into sections.

The code snippet 3.5 was taken from online board game Megamek (Anon., n.d.). Some source code was actually removed in order to make the example presentable, but the entire function consisted of 78 lines of code. This function is not only long, but it clearly has a lot of purposes as well. The reader doesn't have to know much about problem domain to realize that this piece of code handles the tasks that need to be done at the end of a turn, however, it is obvious that there are three distinctive tasks here: *a check for dishonored enemies, an update to fleeing entities and another check for broken enemies after.* On top of that, there is a mix of abstractions levels with the usage of low level system functions such as *StringBuilder* and *append* along with the higher level problem domain functions like *getForcedWithdrawal, getEntitiesOwned* and *checkEnemyBroken*. To make matters worse, there is some conditional statement that is duplicated three times. Overall, it would be a nightmare to fully understand this piece of code, let alone enhance it, there are just too many different things going on at several abstraction levels.

```
public void endOfTurn() {

	StringBuilder msg1 = new StringBuilder(
			"Checking for dishonored enemies.");

	// If the Forced Withdrawal rule is not turned on, then it's a fight
	// to the death anyway.
	if (!getForcedWithdrawal()) {
		msg1.append("\n\tForced withdrawal turned off.");
		return;
	}

	for (Entity mine : getEntitiesOwned()) {
	... Code here checks for the enemies that were dishonored during the turn
	}

	StringBuilder msg2 = new StringBuilder(
			"Updating my list of falling back units.");

	if (!getForcedWithdrawal()) {
		msg2.append("\n\tForced withdrawal turned off.");
		return;
	}

	for (Entity mine : getEntitiesOwned()) {
		if (myFleeingEntities.contains(mine.getId())) {
			continue;
		}
		if (wantsToFallBack(mine)) {
			msg2.append("\n\tAdding ").append(mine.getDisplayName());
			myFleeingEntities.add(mine.getId());
		}
	}

	if (!getForcedWithdrawal()) {
		return;
	}

	for (Entity entity : getEnemyEntities()) {
		getHonorUtil().checkEnemyBroken(entity, getForcedWithdrawal());
	}
}
```

Code Snippet 3.5 Megamek - Board.java

After a few tweaks, separating the three distinctive tasks performed, refactoring out the duplicated code into a separate function, encapsulating low level system functions to be reusable, it was possible to reduce this function to three lines long. Of course, the extracted functions are still fairly long, but at least anyone reading the source code will be able to fully comprehend what happens at the end of turn and if needed, they can further check how each of the extracted functions are implemented. It would be hard to find any logical reasons to further decompose the concept of this function, therefore it can be concluded that it complies with single responsibility principle.

```
public void processEndOfTurn() {
    checkForDishonoredEnemies();
    updateMyFleeingEntities();
    checkForBrokenEnemies();
}
```

Code Snippet 3.6 Megamek - Board.java (modified)

3.2.3 Arguments

Function arguments are a big part of any function concept and suggest that the function will revolve around operating on those arguments. Depending on the number of arguments, functions can be divided into following types:

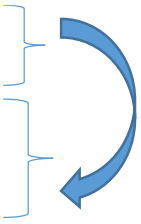
- Niladic – zero arguments,
- Monadic – one argument,
- Dyadic – two arguments,
- Triadic – three arguments,
- Polyadic – more than three arguments.

The more arguments a function has, the harder it is to understand its intention. Also, from a testing point of view, it is increasingly more challenging to account for all possible combinations of arguments to work as they are supposed to, therefore we can conclude that functions should have no more than three arguments, preferably though – none, one or two.

There are a few ways to avoid making functions with large amount of arguments. The most common ones are to either instance some variables instead of passing them around as arguments or to create objects from several arguments. Code snippet 3.7 shows an example taken from graphics framework JHotDraw (Anon., n.d.). There exists *direction* utility function that takes four coordinates as parameters and returns a relative direction between those two points. There is, however, no natural way to realize whether the result is between the first point and the second point or the other way around. In fact, it is obvious that *x1* and *y1* parameters belong together based on their names, so they can be wrapped into a higher abstraction level object as show in second part of code snippet 3.7. *x1* and *y1* were combined into a *Point* object *source*, while *x2* and *y2* were merged into an object *destination*. The function *direction* was mainly used for determining relative direction between rectangles, therefore another helper function *getRectangleCenter* was added to avoid duplication when developing further. Not only does this get rid of an ugly polyadic function which would be a pain to test, but also additional level of clarification was added as to how *direction function works.* It is now obvious that the direction is between *source* and *destination* and not the other way around.

```
static public int direction(int x1, int y1, int x2, int y2);
=============================================================

static public int direction(Point source, Point destination);
static public Point getRectangleCenter(Rectangle rectangle) {
   return new Point(rectangle.x + rectangle.width /2, rectangle.y + rectangle.height/2);
}
```



Code Snippet 3.7 JHotDraw - geom.java (original and modified)

Moreover, the new dyadic *direction* function can now be used in the example given previously in code snippet 3.2. Code snippet 3.8 showcases how much such a small change can affect. With new function taking two *Point* objects as arguments, it was possible to use the helper function *getRectangleCenter* to extract the center points from the two rectangles that would later use *Geom.direction* function. The increase in readability is also massive – not only it completely clarifies the purpose of *direction* function, but also what the *int direction* variable holds, which is the relative direction between *source* and *destination* points.

```
Rectangle rStart = start().owner().displayBox();
Rectangle rEnd = end().owner().displayBox();

int direction = Geom.direction(getRectangleCenterX(rStart), getRectangleCenterY(rStart),
                getRectangleCenterX(rEnd), getRectangleCenterY(rEnd));

private int getRectangleCenterX(Rectangle rectangle){
   return rectangle.x + rectangle.width/2;
}

private int getRectangleCenterY(Rectangle rectangle){
   return rectangle.y + rectangle.height/2;
}
=====================================================================
Point source = Geom.getRectangleCenter(start().owner().displayBox());
Point destination = Geom.getRectangleCenter(end().owner().displayBox());

int direction = Geom.direction(source, destination);
```

Code Snippet 3.8 JHotDraw - ElbowConnection.java (modified)

Another example shown in code snippet 3.9 is taken from a turn-based strategy game FreeCol (Anon., n.d.) where the *addServer* function is used to add a new instance of game server to the existing list of servers. Such polyadic functions will eventually be required to be written, but generally, when a function has so many arguments, it suggests that those arguments are part of the same concept. In fact, in this case, it is fairly obvious that the *addServer* function will operate on the given arguments to form a new instance of server within an existing list. It is then simply better to encapsulate this extensive list of arguments into a separate class and while we still end up a constructor which has the same amount of arguments, the constructor was made private and instead a static function *fromAttributes* was also created to further add to readability by explaining the meaning of the constructor arguments. It made the creation of the *Server* instance clearer by using this static function and further simplified the usage of the *addServer* function since it was transformed from a polyadic eight argument function into a simple monadic one.

```
public synchronized void addServer(String name, String address, int port, int slotsAvailable,
                int currentlyPlaying, boolean isGameStarted, String version, int gameState);
=============================================================================================

public class Server {
        private String name;
        private String address;
        private int port;
        private int slotsAvailable;
        private boolean isGameStarted;
        private String version;
        private int gameState;

        private Server (String name, String address, int port, int slotsAvailable,
                int currentlyPlaying, boolean isGameStarted, String version, int gameState) {
                ...
        }

        public static Server fromAttributes(String name, String address, int port, int
        slotsAvailable, int currentlyPlaying, boolean isGameStarted, String version, int
        gameState){
                return new Server (...);
        }
}
=============================================================================================

Server server = Server.fromAttributes(...);
addServer(server);
```

Code Snippet 3.9 Freecol - Server.java (original and modified)

3.2.4 Duplication

Code duplicated is probably one of the most emphasized 'Clean Code' characteristics. Nearly all practices and principles of software development nowadays have some sort of mention towards code duplication and how bad it is. In fact, one of the goals of object oriented programming is to encapsulate code blocks into separate classes and functions that would otherwise have to be duplicated. Doing so not only has immense affect to code readability but it also allows to make source code highly reusable.

Duplication within code might sometimes seem like an easy and innocent solution, but often it is done without thinking about potential damage to source code maintainability and eventually cost of fixing bugs, especially if duplicated code is at algorithm level. Any time an error is found and it is within a duplicated block of code, then it is multiplied by the amount of times that block was duplicated, because it has to be fixed multiple times. Mostly encountered types of duplication are as follows:

- Type,
- Data,
- Algorithm.

Type duplication is often very hard for developers to notice, let alone fix it. Usually type duplication occurs when several functions are doing seemingly similar operations but on different data types. In this case, generic types can be used to eliminate such duplication as shown in code snippet 3.10.

DUPLICATED CODE

```
public boolean findBoolean(boolean item){
  return (boolean)itemContainer.get(item);
}

public String findString(String item){
  return (String)itemContainer.get(item);
}
```

DUPLICATION REMOVED

```
public T find(T item){
  return (T)itemContainer.get(item);
}
```

Code Snippet 3.10 Custom example of type duplication

Data duplication is fairly easy detect and fix. It usually occurs when a function is developed without the thought of extensibility and then later instead of enhancing the initial function, new ones are added by slightly modifying the initial one. Usually, the most common fix for such duplication is to enhance initial function to take appropriate parameters as shown in code snippet 3.11.

DUPLICATED CODE

```
public Location moveUp(){
   Character character = getCharacter();
   character.move("Up");
   return character.position;
}

public Location moveDown(){
   Character character = getCharacter();
   character.move("Down");
   return character.position;
}

public Location moveLeft(){
   Character character = getCharacter();
   character.move("Left");
   return character.position;
}

public Location moveRight(){
   Character character = getCharacter();
   character.move("Right");
   return character.position;
}
```

DUPLICATION REMOVED

```
public Location move(String direction){
   Character character = getCharacter();
   character.move(direction);
   return character.position;
}
```

Code Snippet 3.11 Custom example of data duplication

Algorithm duplication is quite easy to notice, but probably the hardest to eliminate since it requires understanding of delegates. Code snippet 3.12 highlights such duplication and it is worth noting that the structure of the functions is similar to the behavior design pattern - *Template Method* (WikiPedia, n.d.). In fact, the logic behind this design pattern architecture is the same logic used to solve algorithm type code duplication – replacing select steps of an algorithm without altering the base structure.

DUPLICATED CODE

```
public void makeHamSandwich(){
        getBread();
        applyButter();
        applyHam();
        applySallad();
}

public void makeTurkeySandwich(){
        getBread();
        applyButter();
        applyTurkey();
        applySallad();
}
```

DUPLICATION REMOVED

```
public void makeSandwich(iFilling filling){
        getBread();
        applyButter();
        filling.apply();
        applySallad();
}
```

Code Snippet 3.12 Custom example of algorithm duplication

3.3 Comments

It would be hard to imagine source code without comments. Programming languages are just not expressive enough sometimes for developers to convey the intention of their code. Commenting is essential part of software development, especially in larger projects. A well placed comment can save an enormous amount of time for the reader. On the other hand, comments can also be completely redundant and litter the code to become hardly readable.

The general guideline suggested by Robert C. Martin is to avoid writing comments as much as possible. He argues that almost always it is possible to write expressive enough code removing the need for comments.

```
/*
 *  This file is part of the Heritrix web crawler (crawler.archive.org).
 *
 *  Licensed to the Internet Archive (IA) by one or more individual
 *  contributors.
 *
 *  The IA licenses this file to You under the Apache License, Version 2.0
 *  (the "License"); you may not use this file except in compliance with
 *  the License.  You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 *  Unless required by applicable law or agreed to in writing, software
 *  distributed under the License is distributed on an "AS IS" BASIS,
 *  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 *  See the License for the specific language governing permissions and
 *  limitations under the License.
 */
```

Code Snippet 3.13 Heritrix - general example of legal comments

3.3.1 Good Comments

There are only few occasions where commenting can be justified. Legal comments are one of them. In nearly every larger project, legal comments are used to state copyright and give credit to the source code authors. Luckily, IDE's nowadays are smart enough to hide these kind of comments so they don't become just another nuisance for developers.

Very rarely comments can also be extremely useful by providing explanation to ideas behind certain design. Code snippet 3.14 shows an example where such comment is really well placed and explains what the users that are going to be extending the class should do if they don't want certain effects.

```
/**
 * Called when a CrawlURI is ruled out of scope. Override if you don't want
 * logs as coming from this class.
 */
protected void outOfScope(CrawlURI caUri) {
	if (fileLogger != null) {
		fileLogger.info("REJECT " + caUri);
	}
}
```

Code Snippet 3.14 Heritrix - Crawler.java

Another occasion where commenting is used (and sometimes overused) is documentation of public API's. It is, however, it is worth noting that not every public function or variable should have documentation comments in source code. Commenting public APIs should be done moderately where source code is not enough to understand the intent behind it.

3.3.2 Bad Comments

Unfortunately, most comments are not good and instead of being useful, they simply clutter up the screen with redundant or even misleading information. Code snippet 3.15 showcases a perfect example, where the comment was used ("add a new node:") at some point to explain what the code was doing, but that code was later also commented out and all that was left – three lines of nothing. Not only is the comment "add a new node:" already redundant since it is obvious what the code was doing, but it was never maintained. In general, such commented blocks should just simply be removed from the source code as they can leave the reader wondering if that part of the code is somehow essential and is commented out for a reason. Suffice to say, such comments can certainly become a source of confusion, especially when trying to enhance existing code.

```
// add a new node:
// MindMapNode newNode=((ControllerAdapter)getController()).newNode();

// ((MapAdapter) getMap()).insertNodeInto(newNode, getNode(), 0);
```

Code Snippet 3.15 Freemind - MapModuleManager.java

Redundant comments are the most common, they simply repeat what is already written in the code without actually adding any additional clarification. Code snippet 3.16 shows an example where a comment is completely redundant – it simply says that the following function is a constructor.

```java
/**
* Constructor.
*/
public Scoper() {
    super();
}
```

Code Snippet 3.16 Heritrix - Scoper.java

Even worse, over documented public API's can extremely clutter up the code. A champion of redundancy can be seen in Code snippet 3.17. Such commenting can take a huge part of the computer screen and severely impair the reader's ability to navigate the source code seamlessly, therefore damaging readability and understandability.

```java
/**
 * Returns the bean descriptor.
 *
 * @return The bean descriptor.
**/
 public BeanDescriptor getBeanDescriptor ()
{
    return new BeanDescriptor (beanClass_);
}

 /**
 * Returns the descriptors for all events.
 * @return The descriptors for all events.
 **/
 public EventSetDescriptor[] getEventSetDescriptors()
{
    return events_;
}

 /**
  * Returns the descriptors for all properties.
  * @return The descriptors for all properties.
  **/
  public PropertyDescriptor[] getPropertyDescriptors()
 {
     return properties_;
 }
```

Code Snippet 3.17 JTOpen - DataDescriptor.java

Comments are written along with source code for a reason – they are to be read by people who are going to be also reading and possibly modifying the source code. Using formatting tags in comments completely defeats that purpose. Any tool that is going to be extracting comments from the source code should also be able to appropriately format them.

```
/**
 * <P>
 * VRunJavaApplication has three parameters command line parameters.  All
 * parameters are optional.  The parameters are:
 * <UL>
 *   <li><i>System</i> - the system that contains the Java program
 *   <li><i>UserID</i> - run the Java program under this userid
 *   <li><i>Password</i> - the password for the userid
 * </UL>
 *
 * <P>
 * For example, to run Java programs on system "mySystem":
 * <a name="ex"> </a>
 * <PRE>
 * java utilities.VRunJavaApplication mySystem
 * </PRE>
 *
 * <P>
 * See the javadoc for Toolbox classes VJavaApplicationCall and
 * {@link com.ibm.as400.access.JavaApplicationCall JavaApplicationCall}
 * for a list of commands to run once the program is started.
 *
**/
```

Code Snippet 3.18 JTOpen - RunJavaApplication.java

In general, there is a fine line between good and bad comments in source code. There exist a lot of cases where different developers would argue about the usefulness of certain comments, therefore it would be hard to define absolute best practice to write comments.

3.4 Formatting

Source code formatting has to be considered as one of the 'Clean Code' characteristics simply because it is one of the biggest contributors when it comes to readability. Consistently applied formatting rules allow developer teams to instinctively recognize blocks of code that belong to each other within classes or functions as well as seamlessly read the code and easily find relevant information. Source code is most often read left to right and from top to bottom. In general, writing source code is mainly about solving problems and understanding how the problem was solved, therefore it is extremely important for the authors of the code to explain how they solved certain problems.

3.4.1 Vertical formatting

Vertical formatting is all about organization of source code files and how it helps the readers of the code to comprehend that code. The name of the file should tell if the reader is in the correct place and where the relevant information is going to be, followed by high abstraction level functions that define concepts and algorithms. Towards the end, the level of detail will increase, allowing the reader to see lower level functions. The following principles should be followed in order to keep vertical formatting clean:

- Openness – lines of code that belong together or form a certain concept should be clustered into blocks and those blocks can then be separated by empty lines. This is always the case for different functions within the code, but sometimes such blocks can be found within the functions themselves and should also be separated by empty lines. Code snippet 3.19 showcases how important openness is and how much harder it is read the code if this principle is not followed.

NO VERTICAL OPENNESS

```
private static class BaseBean {
        private final Object p2;
        private final Object x1;
        public BaseBean(final Object p2,
                        final Object x1) {
                this.p2 = p2;
                this.x1 = x1;}
        public Object getP2() {
                return p2;}
        public Object getX1() {
                return x1;}
        public String toString() {
                return "I am bean.";}
}
```

EMPTY LINES BETWEEN FUNCTIONS

```
private static class BaseBean {

        private final Object p2;
        private final Object x1;

        public BaseBean(final Object p2,
                        final Object x1) {
                this.p2 = p2;
                this.x1 = x1;
        }

        public Object getP2() {
                return p2;
        }

        public Object getX1() {
                return x1;
        }

        public String toString() {
                return "I am bean.";
        }
}
```

Code Snippet 3.19 Log4J - BaseBean.java (original and modified)

- Density – the opposite of openness, having relevant lines or blocks of code close to each other is equally important. Usually this principle is broken with the abusive usage of Javadocs comments. It can be seen in code snippet 3.20 how the association between relevant functions and variables can be depreciated with the usage of useless comments.
- Distance – this is especially relevant in large classes and functions, the separation between variable declaration and their usage or between functions that are dependent on each other can make a huge difference when trying to determine how exactly they are operated on. Even though IDEs have made it fairly easy to find the references to the usage of variables or functions, it is never fun to keep jumping from one end of a file to the other end when trying to fix bugs. Keeping the dependent functions and variables close to each other will increase readability.
- Ordering - it is natural to expect the highest level abstraction functions and those that are least dependent to be at the top of the file. It can be confusing if the function that gets called by another function is the first one read, because then we might be reading details that we don't actually need, thus preventing us from quickly glancing over source files and finding the right information.

WITH USELESS COMMENTS

```
public class HistoryBrowser{

    /**
     * The history browser.
     */
    protected HistoryBrowser historyBrowser;

    /**
     * Gets the history browser.
     *
     * @return the historyBrowser
     */
    public HistoryBrowser getHistoryBrowser() {
        return historyBrowser;
    }

    /**
     * Adds the NodeInsertedCommand to historyBrowser.
     *
     * @param newParent
     *            New parent node
     * @param newSibling
     *            New (next) sibling node
     * @param contextNode
     *            The node to be appended
     */
    public void nodeInserted(Node newParent, Node newSibling, Node contextNode) {
        historyBrowser.addCommand(createNodeInsertedCommand(newParent,
                newSibling, contextNode));
    }

}
```

COMMENTS REMOVED

```
public class HistoryBrowser{

    protected HistoryBrowser historyBrowser;

    public HistoryBrowser getHistoryBrowser() {
        return historyBrowser;
    }

    public void nodeInserted(Node newParent, Node newSibling, Node contextNode) {
        historyBrowser.addCommand(createNodeInsertedCommand(newParent,
                newSibling, contextNode));
    }
}
```

Code Snippet 3.20 Batik - HistoryBrowser.java (original and modified)

3.4.2 Horizontal formatting

Horizontal width is just as important as vertical size. Horizontal space is used to emphasize things that are related to each other, just like in vertical formatting. Similar principles are therefore used. Code

snippet 3.21 shows source code taken from Java software development tool Quilt (Anon., n.d.) where all formatting was removed. Without formatting, the piece of code is completely unreadable. It would require a lot of effort to understand what the code does. This code will be used as an example to show how the usage horizontal formatting, and in turn vertical formatting principles, can affect readability.

ALL FORMATTING REMOVED

```
public class ComplexConnector extends Connector {
private Edge edge;private Edge[] edges; private Vertex source;
private void checkTarget(final Vertex target){if(target == null){
throw new IllegalArgumentException("target may not be null");}
if(source.getGraph()!=target.getGraph()){throw new IllegalArgumentException(
"ComplexConnector's target must be in the same graph");}}
private void rangeCheck(int n){if(n<0||n>=edges.length){throw new IllegalArgumentException (
"ComplexConnector index "+n+" out of range 0.."+(edges.length-1));}}
public void setTarget(Vertex v,int n){checkTarget(v);rangeCheck(n);edges[n].setTarget(v);}}
```

Code Snippet 3.21 Quilt - ComplexConnector.java (modified)

- Openness – this is used mostly in assignment statements to make separation obvious. Also, there shouldn't be more than one statement per line, but applying this principle has a side effect (not a bad one) of increasing vertical size.

HORIZONTAL OPENNESS APPLIED

```
public class ComplexConnector extends Connector {
private Edge edge;
private Edge[] edges;
private Vertex source;
private void checkTarget (final Vertex target){
if(target == null){
throw new IllegalArgumentException ("target may not be null");
}
if(source.getGraph() != target.getGraph())
{
throw new IllegalArgumentException ("ComplexConnector's target must be in the same graph");
}
}
private void rangeCheck (int n){
if(n < 0|| n >= edges.length)
{
throw new IllegalArgumentException ("ComplexConnector index " + n + " out of range 0.." +
(edges.length - 1));}}
public void setTarget (Vertex v, int n){
checkTarget(v);
rangeCheck(n);
edges[n].setTarget(v);
}
}
```

Code Snippet 3.22 Quilt - ComplexConnector.java (modified)

- Density – notice a space between functions *checkTarget, rangeCheck* and *setTarget* and their arguments. This shouldn't be the case because both function name and its arguments are part of one concept and therefore should be denser. Also density can sometimes be used in conjunction with

openness to accentuate different operators by keeping them close. However, this particular principle can be argued against and should only be used if the entire developer team agrees so that the consistency is maintained.

HORIZONTAL DENSITY APPLIED

```
public class ComplexConnector extends Connector {
private Edge edge;
private Edge[] edges;
private Vertex source;
private void checkTarget(final Vertex target){
if(target == null){
throw new IllegalArgumentException("target may not be null");
}
if(source.getGraph() != target.getGraph())
{
throw new IllegalArgumentException("ComplexConnector's target must be in the same graph");
}
}
private void rangeCheck(int n){
if(n<0|| n>=edges.length) //operators are different, therefore density is used
{
throw new IllegalArgumentException("ComplexConnector index " + n + " out of range 0.." +
(edges.length - 1));}}
public void setTarget(Vertex v, int n){
checkTarget(v);
rangeCheck(n);
edges[n].setTarget(v);
}
}
```

Code Snippet 3.23 Quilt - ComplexConnector.java (modified)

- Alignment – this is a principle that was especially popular in older programming languages, but is still sometimes used. The main idea is to showcase the declarations in a set of aligned structures. Many developers have moved on from using this principle since it fails to accomplish much in terms of readability and can sometimes be slightly misleading (especially when assignment operators differ).

HORIZONTAL ALIGNMENT APPLIED

```
public class ComplexConnector extends Connector {
private Edge     edge;
private Edge[]   edges;
private Vertex   source;
public ComplexConnector(Edge edge){
this.edge     =   edge;
this.edges    =   null;
this.source   =   null;
}
...
}
```

Code Snippet 3.24 Quilt - ComplexConnector.java (modified)

- Identation – the most important horizontal formatting principle that allows readers to easily realize the structure of the classes, nesting levels and scopes. Thankfully, IDEs nowadays are powerful enough to help relieve this burden from developers with automatic indentation tools.

```
HORIZONTAL INDENTATION APPLIED
public class ComplexConnector extends Connector {
        private Edge edge;
        private Edge[] edges;
        private Vertex source;

        private void checkTarget(final Vertex target) {
                if (target == null) {
                        throw new IllegalArgumentException("target may not be null");
                }
                if (source.getGraph() != target.getGraph()) {
                        throw new IllegalArgumentException(
                                        "ComplexConnector's target must be in the same graph");
                }
        }

        private void rangeCheck(int n) {
                if (n<0 || n>=edges.length) {
                        throw new IllegalArgumentException("ComplexConnector index " + n
                                        + " out of range 0.." + (edges.length - 1));
                }
        }

        public void setTarget(Vertex v, int n) {
                checkTarget(v);
                rangeCheck(n);
                edges[n].setTarget(v);
        }
}
```

Code Snippet 3.25 Quilt - ComplexConnector.java (original)

3.5 Objects and data structures

Objects allow developers to encapsulate certain data, but expose the behavior while data structures usually don't have any significant behavior but they expose the data. The two structures offer different features: using objects allows easy addition of new objects without affecting behavior while using data structures enables developers to add new behaviors to operate on that data. Depending on the problem being solved, and whether flexibility to add new types of data or different behaviors is needed, the appropriate method should be used.

Consider the following example: code snippet 3.26 shows an implementation taken from the extensible graphics framework JHotDraw (Anon., n.d.) where the *Figure* interface is used to define the base set of methods that a figure should have while code snippet 3.27 is an alternative solution to the same problem but data structures are used instead and all behavior is in a separate *FigureTools* class. In the current form, both solutions achieve exactly the same with roughly the same amount of code, therefore they're pretty much equally good. However, when designing such system, the following question must be answered in order to determine whether use with data structures or objects – what is going to be added more often: types of figures or operations? If more types of figures are going to be added, then obviously the object oriented solution is quite superior since each new type of figure would require changing each operation in *FigureTools* class while in the object oriented solution nothing would be changed and only an additional class needs to be added. On the other hand, if more operations are to be added, then the solution using

data structures is better, because the object oriented solution would require revisiting each *Figure* interface implementation and appending them with appropriate code to deal with those new operations.

SOLUTION USINGS OBJECTS

```
public interface Figure {

        public void displayBox(Point origin, Point corner);
}

public class EllipseFigure implements Figure {

    private Rectangle fDisplayBox;

    public basicDisplayBox(Point origin, Point corner) {
        fDisplayBox = new Rectangle(origin);
        fDisplayBox.add(corner);
    }

    public Rectangle getDisplayBox(){
       return fDisplayBox;
    }
}

public class BatteryFigure implements Figure {

    private Rectangle fDisplayBox;

    public basicDisplayBox(Point origin, Point corner) {
        fDisplayBox = new Rectangle(origin);
        fDisplayBox.height = 20;
        fDisplayBox.width = 30;
    }

    public Rectangle getDisplayBox(){
       return fDisplayBox;
    }
}
```

Code Snippet 3.26 JHotDraw - Figure.java, EllipseFigure.java, BatteryFigure.java (original)

In this case, the authors of JHotDraw (Anon., n.d.) were designing a framework where custom figures are going to be more likely additions when extending, therefore they went with an object oriented solution as opposed to using data structures. Data structures result in procedural code, which seems to be fading away from the industry of software development. There is no doubt that object oriented solutions will be superior for solving most problems, but procedural code should not be forgotten as there are definitely cases when it is appropriate to use it.

SOLUTION USING DATA STRUCTURES

```
public class EllipseFigure {

    public Point origin;
    public int majorAxis;
    public int minorAxis;
}

public class BatteryFigure {

    public Point origin;
    public int height;
    public int width;
}

public class FigureTools {

       public Rectangle getFigureDisplayBox(Object figure) throws InvalidFigureException {
           if (figure instanceof EllipseFigure){
              EllipseFigure ellipse = (EllipseFigure)figure;
              Rectangle figureDisplayBox = new Rectangle(ellipse.origin);
              figureDisplayBox.height =  ellipse.minorAxis;
              figureDisplayBox.width = ellipse.majorAxis;
           }
           else if (figure instanceof BatteryFigure){
              BatteryFigure battery = (BatteryFigure)figure;
              Rectangle figureDisplayBox = new Rectangle(battery.origin);
              figureDisplayBox.height =  battery.height;
              figureDisplayBox.width = battery.width;
           }
           throw new InvalidFigureException();
       }
}
```

Code Snippet 3.27 JHotDraw - FigureTools.java, EllipseFigure.java, BatteryFigure.java (modified)

3.6 Error Handling

Things don't always go as expected, in fact, most of the time they go wrong. The same is applicable to source code. It is absolutely vital for software to be equipped with tools to be able to handle situations when errors occur and take appropriate course of action. At the same time it is important to separate the logic of error handling and actual problem solved within code. Key principles to follow in order to achieve appropriate error handling are as follows:

- Exceptions should be favored over error codes. The main issue with using error codes is the inability to separate the logic of error handling from the rest of the code. It ends up filling the source code with mixed information, therefore it is better to use exceptions when an error is encountered and have exception handling in separate methods.
- Do not use checked exceptions. Most languages don't even have them. One of the main reasons to not use checked exceptions is because they instantly break Open/Closed principle which says that new functionality should result in minimum required changes for the existing code. SOLID (WikiPedia, n.d.) principles are agreed by most software experts to be followed and therefore any technique that opposes them should only be used with caution.
- Exceptions should have informative error messages.

- Bundle up exceptions by identifying commonalities between them. Often exceptions can be classified by their source or type. Using wrapper classes for such occasions is particularly useful because it can help minimize reliance on third-party APIs.
- Returning or passing *null* is bad. *Null* is not only one of the main sources of errors, but also it can clutter the code with checks for *null* for no reason. It is simply better to either throw exceptions or use empty objects as opposed to using *null*.

3.7 Unit tests

Testing is a really important part of software development lifecycle. Code that is written for testing purposes should be kept clean because it can affect readability and maintainability just as much. Test Driven Development is very popular nowadays and many developers embrace it because it forces writing unit tests first and then writing production code. It is a cycle that ties testing and production code so that both are equally maintained.

Unit tests enable production code to be flexible and reusable without fear that some tweaks to the code will break it. The main thing that affects cleanliness in tests is readability. In order to write clean and readable unit tests, the following principles should be followed:

- Unit tests should contain a clear structure. The basic structure followed should include setting up conditions required for testing, calling the methods or triggers being tested and verifying that the results are as expected. This way all the details are separated into their appropriate functions at a lower abstraction level and anyone who is reading the test will be able to quickly understand the purpose of the test and how it works. Code snippet 3.28 shows an example taken from FindBugs (Anon., n.d.) where such clear structure is used.

```
@Test
public void testAllRegressionFilesJavac() throws IOException, InterruptedException {
    setUpEngine("build/classes/");

    engine.execute();

    // If there are zero bugs, then something's wrong
    assertFalse("No bugs were reported. Something is wrong with the configuration",
                bugReporter.getBugCollection().getCollection().isEmpty());
}
```

Code Snippet 3.28 FindBugs - RegressionTests.java

- Creating utilities and tools helps to write tests faster. The previous example in figure# also showed that the utility function used *setUpEngine*. Any other test that would follow a similar structure and require the setting up of engine for its initialization could reuse this function.
- Minimum number of asserts per test. In fact, every test should preferably only contain one assert, however, sometimes it would result in a lot of duplicate code in exchange for a minor increase in readability, granted that the tests would have appropriate naming. It becomes a bigger problem if there are several concepts being tested at the same time. In that case, they should definitely be split into different test functions.
- Tests should be fast. There is nothing worse when you have slow tests, because the reluctance to run slow tests significantly decreases the ability to write production code and more importantly, to change existing code.

3.8 Classes

In object oriented programming, classes are at the top of organization hierarchy. They are the largest unit of source and they generally host a dependent set of functions and variables. Nearly every programming language follows a convention to declare variables first starting with public static constants, followed private static variables and private instance variables. Public variables are very rare, because most of the time public variables are replaced by a private ones with a getter function to make them accessible. This list of variables is followed by functions with public functions being first and then private ones. It is worth noting, however, that sometimes it is better to list certain private functions just after the public functions if they are dependent on each other so that the source code can be read from top to bottom.

3.8.1 Size

Just like with functions, smaller classes are preferred, however, it would be hard to measure classes in terms of lines of code. Classes, however, should follow the single responsibility principle and therefore the amount of responsibilities that a class has can be used as a size metric. If the single responsibility principle is embraced for class design, then it can be derived that any class with more than one responsibility is already too large.

Code snippet 3.29 shows a class taken from GeoTools (Anon., n.d.). This class contains 86 public functions. Now that is a large amount of functions for a single class to have. Such classes are bound to break the single responsibility principle and this one is no exception.

```
public class CircularRing extends LinearRing implements
               SingleCurvedGeometry<LinearRing>, CurvedRing {

        public int getNumArcs();
        public boolean contains(Geometry g);
        public boolean covers(Geometry g);
        public void apply(GeometryFilter filter);
        public double getTolerance();
        public boolean touches(Geometry g);
        public double[] getControlPoints();
        public int getDimension();
        public boolean crosses(Geometry g);
        public boolean within(Geometry g);
        public int getBoundaryDimension();
        public int getNumGeometries();
        public Point getInteriorPoint();
        public Point getStartPoint();
        public Point getEndPoint();
        public double getLength();
        public void apply(CoordinateFilter filter);
        public boolean overlaps(Geometry g);
        public void apply(GeometryComponentFilter filter);
        public boolean disjoint(Geometry g);
        public boolean intersects(Geometry g);
        public void apply(CoordinateSequenceFilter filter);
        public String getGeometryType();
        ...
}
```

Code Snippet 3.29 GeoTools - CircularRing.java (original)

Even if we were to theoretically remove 90% of those functions and would be left with the class shown in code snippet 3.30, it would still break single responsibility principle. In this case, *CircularRing* class allows

access to some information about geometric figure's spatial properties as well as providing filtering tools and some algebraic "check" function. With a simple glance at the description, it was possible to identify three different responsibilities in the class with 4 functions.

```
public class CircularRing extends LinearRing implements
               SingleCurvedGeometry<LinearRing>, CurvedRing {

       public Point getStartPoint();
       public Point getEndPoint();
       public void apply(CoordinateFilter filter);
       public boolean disjoint(Geometry g);
}
```

Code Snippet 3.30 GeoTools - CircularRing.java (modified)

Code snippet 3.31 shows the extra responsibilities extracted into their own classes. In fact, initial *CircularRing* class was a derivative from *LinearRing* and each of its subclasses actually have the same three responsibilities with a ton of functions, so by adding these extra classes we would remove a lot of duplication in other classes.

```
public class SpatialProperties {
       public int getNumGeometries();
       public Point getInteriorPoint();
       public Point getStartPoint();
       public Point getEndPoint();
       ...
}

public class AlgebraicFunctions{
       public boolean overlaps(Geometry g);
       public boolean covers(Geometry g);
       ...
}
```

Code Snippet 3.31 Custom classes derived from GeoTools – CircularRing.java

3.8.2 Open-Closed

In SOLID (WikiPedia, n.d.) principles, the single responsibility principle is followed by the open-closed principle which states that "software entities should be open for extension, but closed for modification" (WikiPedia, n.d.). It essentially means that if there is a need to enhance classes (applies for modules and functions as well), it should be done without affecting existing ones.

Code snippet 3.32 shows one of the classes that was extracted as an extra responsibility class from the previous example. Any time there was a need to add a new algebraic function, we would be required to modify the existing *AlgebraicFunctions* class. This is a clear violation of the open-closed principle. In fact, to some extent, the single responsibility principle is violated here as well, because it has more than one reason to change: any time a new function has to be added and when an algorithm of an existing function changes.

```
public class AlgebraicFunctions{
	public boolean disjoint(Geometry g);
	public boolean touches(Geometry g);
	public boolean intersects(Geometry g);
	public boolean contains(Geometry g);
	public boolean overlaps(Geometry g);
	public boolean covers(Geometry g);
}
```

Code Snippet 3.32 Custom class derived from GeoTools – CircularRing.java

In order to comply with the open-closed principle, there should be an abstract "hotspot" class that is designed for an easy extension and every function should be declared in its own subclass as shown in code snippet 3.33. With such a design, any time a new algebraic function needs to be added, it can be done via subclassing instead of modifying the existing classes, thus not only fully achieving single responsibility principle, but also open-closed principle.

```
abstract public class AlgebraicFunction {
	public AlgebraicProperty(Geometry g);
	abstract public boolean check();
}

public class AlgebraicCrosses extends AlgebraicFunction {
	public AlgebraicCrosses(Geometry g);
	@Override
	public boolean check();
}

public class AlgebraicDisjoint extends AlgebraicFunction {
	public AlgebraicDisjoint(Geometry g);
	@Override
	public boolean check();
}

...
```

Code Snippet 3.33 Custom classes derived from GeoTools – CircularRing.java

3.9 Systems

If all the 'Clean Code' characteristics mentioned so far were followed to the smallest detail when building parts of system, would it make entire system clean? The answer is no. Even if every individual component is as clean as it can be, at system abstraction level, there has to be organization as well. There exist an enormous amount of material already produced with guidelines as to how systems should be designed and implemented. The main commonality that can be extracted from already existing techniques and principles is the mention of separation of concerns, which "is a design principle for separating a computer program into distinct sections, such that each section addresses a separate concern" (WikiPedia, n.d.). An architectural pattern model-view-controller (MVC) was invented so that separation of concerns principle would be followed. "It divides a given software application into three interconnected parts, so as to

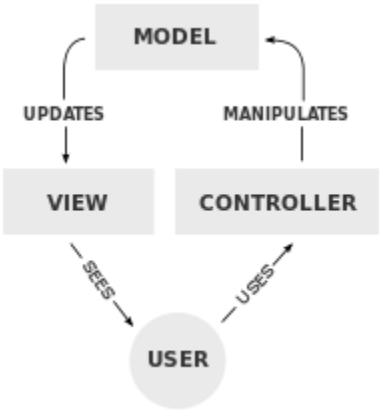


Figure 3.1 Model-View-Controller flow chart

separate internal representations of information from the ways that information is presented to or accepted from the user" (WikiPedia, n.d.).

Languages like C# or Ruby even have partial classes so that the separation of concerns can be enforced. Different aspects of the same class can be declared in separate files to make it easier to enhance the code with new functionality. In general, systems should be just as clean as its individual parts and clearly convey the intent, otherwise maintainability and quality usually pays the cost of unclean system.

It is worth noting, that by no means, all 'Clean Code' characteristics have been mentioned in this chapter. When it comes to fairly abstract units of source code such as classes, groups of developers tend to have their own and more precise techniques of how to write clean classes.

CHAPTER 4: ANALYSIS TOOL

The study includes development of the analysis tool that can be used to evaluate the extent to which open source systems satisfy the criteria of 'Clean Code'. The nature of the tool suggests that it will attach itself to a set of data, parse it and return the results based on the criteria specified.

The analysis tool parses the source code of a project. There exist a variety of libraries that offer the functionality of analyzing source code. For this study, JDT library will be used as the code will only be analyzed statically, and not during runtime. It offers a set of functions for analysis and manipulation of Java class files. More specifically, it offers symbolic information of the given class in a form of a class object which contains methods, fields, identifiers, etc.

The challenging part of this analysis tool is to actually determine whether the criteria of 'Clean Code' has been met. For example, is the method name appropriate? Or, did the function meet its purpose? Because the analysis tool needs to be able to analyze the data based on the set of characteristics identified, these two parts of the study are closely related and it also introduces several limitations. The characteristics that the analysis tool are using as criteria need to be simple enough to determine within the code. For example, function abstraction level detection would be nearly impossible to develop within the analysis tool given the limited timeframe of the study.

4.1 Abstract syntax tree

The Abstract Syntax Tree is the base framework for many powerful tools. The Abstract Syntax Tree maps plain source code in a tree form. This tree is more convenient and reliable to analyze and modify programmatically than text-based source (Kuhn, 2006).

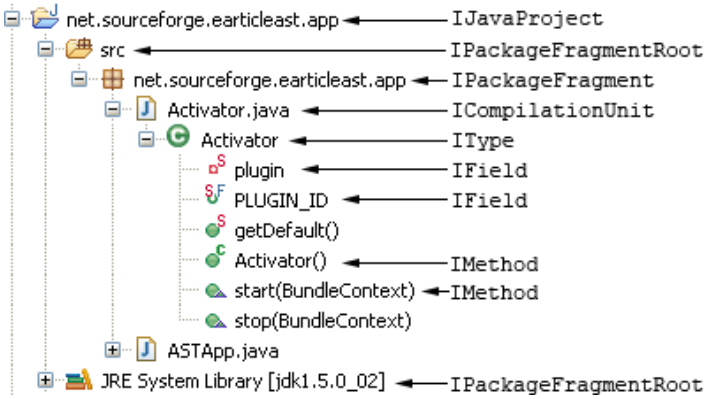


Figure 4.1. Java model overview

An abstract Syntax Tree is the way that IDE's looks at the source code: every source file is entirely represented as tree of AST nodes. These nodes are all subclasses of *ASTNode*. Every subclass is specialized for an element of the certain Programming Language. E.g. there are nodes for method declarations (*MethodDeclaration*), variable declaration (*VariableDeclarationFragment*), classes and assignments and so on. One very frequently used node is *SimpleName*. A *SimpleName* is any string of source that is not a keyword, a *Boolean* literal (true or false) or the *null* literal. All AST-relevant classes are located in the package *org.eclipse.jdt.core.dom* of the *org.eclipse.jdt.core* plug-in. It is also worth noting that there exists an AST Viewer plugin that can display the AST for specified source code in a tree form, which is a big help if someone wants to visualize of what an AST looks like.

Figure 4.1 shows an example of how different source code units (projects, packages, classes, functions, etc.) are seen by the IDEs. In this case, the image is from Eclipse IDE. It is worth noting, that this is not an AST yet. In fact, all those units (*IPackacgeFragment, ICompilationUnit, IType, IMethod*) are interfaces that can be implemented and those implementation classes would then be associated with a certain amount of source code. For example, the *ICompilationUnit* derivative can be associated with all the source code

that is in the *Activator.java* file while *IType* implementation can be associated with only the source code of *Activator* class and thus if there was another class declared in the same file, it would be associated with a different derivative of *IType*. These derived classes can then be used by JDT's *ASTParser* in order to build an AST from a given block of source code, whether it is a class, a function or only a single variable.

Figure 4.2 showcases the workflow of an application using AST. Each step corresponds to the following:

1. Java source code can be parsed from a Java file or the editor within Eclipse directly.
2. JDT provides a parser located in a plugin org.eclipse.jdt.core.dom.ASTParser which forms an AST.
3. The Abstract Syntax Tree is the result of previous step. It is a direct reflection of a given source code in a tree model.
4. AST can then be manipulated in two ways: by directly modifying the AST or by noting the modifications in a separate protocol. This protocol is handled by an instance of ASTRewrite. JDT provides the API for AST manipulation which contains features, such as new node creation, replacement of existing node, removal of a node, etc.
5. Once changes have been made the AST is submitted back to the parser and reversed back into the source the code.
6. IDocument is a wrapper for the source code that represents new document with the applied changes.

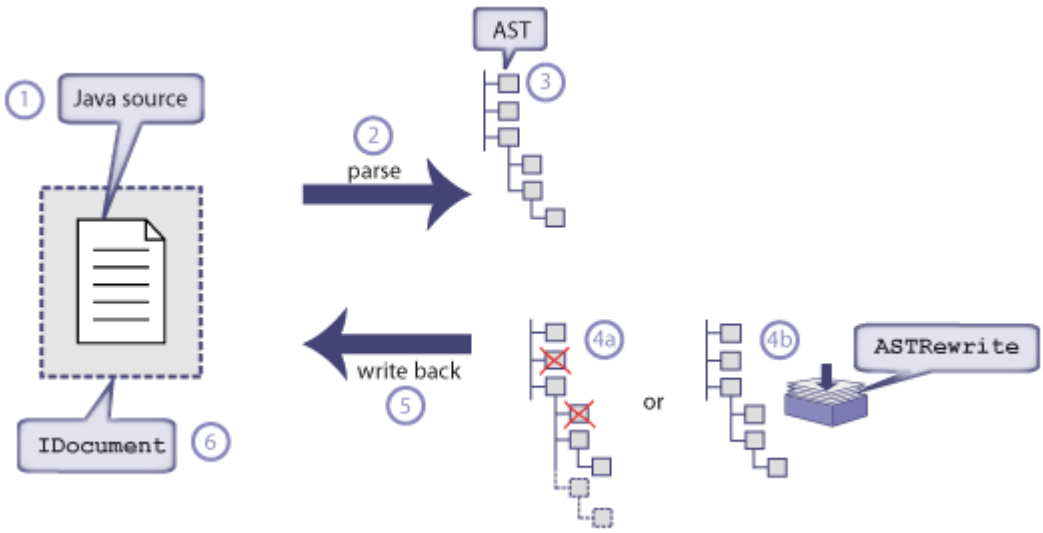


Figure 4.2 Workflow of an application using AST

Due to the nature of this study, in step 1 the analysis tool only parses source code from Java files as it is simply much easier to have the systems being analyzed in separate JAR containers rather than attempting to import each system as a separate project into Eclipse or other IDE. Moreover, no modifications of AST are required as the entire analysis is done statically by using visitor design pattern, therefore step 4-6 do not play any role in the analysis tool. The visitor pattern allows traversal of the AST and to enhance the classes visited by adding additional functions without actually changing the classes themselves. Such additional functionality can then be used in order to collect relevant data about the visited classes and its

contents. It is worth noting that the visitor pattern for AST traversal is already fully implemented in JDT library.

4.2 Calculating the totals

Raw JDT classes cover over 40 node types, each containing various properties about different source code units. Luckily, it has some convenient groupings marked by abstract superclasses:

- Expressions
- Names
- Statements
- Types
- Type Body Declarations

In order to inspect various elements of the code, appropriate functions from JDT's implementation of AST visitor needs to be overridden and then source analysis can be performed.

Counting up the total amount of statements, functions or classes becomes fairly simple as it is simply incrementing some global variable each time an appropriate node (variable declaration, function declaration or class declaration node) was visited. The average amount of statements per function or functions per class can then be derived from the totals.

4.3 Detecting names

Once again, not particularly challenging and using the same logic as for calculating the totals, *VariableDeclarationFragment, ClassDeclaration* or *MethodDeclaration* node has to be visited and within it there is always another nested *SimpleName* node that contains information about the name of the variable, class or function being declared. The acquired name can then be evaluated for required criteria, such as whether it is a single character name, how many characters it has, etc.

4.4 Measuring complexity

The complexity measure was implemented using the guidelines of an already existing tool called GMetrics (Anon., n.d.) that calculates various metrics including complexity for Groovy source code. Code snippet 4.1 shows an example of how complexity can be calculated for a given function. Total complexity is incremented by 1 any time one of the following types of control flow statements have been used:

- *case* statement
- *if* statement
- *for* statement
- *throw* statement
- *while* statement
- *catch* statement
- *return* statement (except if it the last one in the function, then complexity is not affected)
- *?* operator
- *&&* and // logical operations

```
public void visitExample(MethodDeclaration node){                // +1
    if(node.isNotValid()){                                       // +1
         return;                                                 // +1
    }
    node.AttachToParent();
    while(node.isAttached() && node.getSimpleName().exists()){   // +2
         processSimpleNameNode(node.getSimpleName());
    }
    return;
}
```

Code Snippet 4.1 Custom example of complexity metric calculation

Notice the *+1* at the start of the function. Even though it was not one of the criterias that increment complexity, the reason why it is shown in the example is because the default complexity of a function without any control flow statements is 1 since there is a single possible path in the code. Because in the analysis tool developed for this study, overall complexity for class was measured, it was more convenient to start at 0 and increment it by 1 not only for each control flow statement, but also for each function found.

4.5 Comments and duplication

Comments and duplication were measured using a 3rd party tool called SourceMonitor (Campwood, n.d.) that provides static analysis of source code and measures source code metrics.

The SourceMonitor tool provides the ability to omit several types of comments, such as legal comments or TODO comments. Legal comments are detected by assuming that their position is usually at the top of the file. TODO comments are detected using regular expressions with the keyword *TODO.* In this study, only legal comments were omitted from analysis as it is generally accepted and sometimes even necessary to have them.

Code duplication in SourceMonitor is detected by using a fairly simple algorithm that attempts to identify similar sequences of strings while omitting any comments and whitespaces in between. Unfortunately, while the performance of this algorithm is decent, the main issue is that function boundaries are not taken into account. Such drawback can potentially result in false positives. A much better approach would be to use AST based duplication detection, but tools that use such algorithm are not free.

4.6 Limitations

The following obstacles preventing from measuring more metrics were identified during the development of analysis tool:

- Analyzing semantics of comments or names would be desirable for such study, but there aren't any tools available yet with such functionality and the research in this area is lacking suggesting that it would be quite hard to implement any semantical measurements.
- Measuring responsibilities programmatically would be more appropriate size metric than for classes and functions than the amount of statements, however, no research could be found in how to do so.
- Detecting abstraction levels seems to also be in the early stages of research (see related work section) and would be too difficult to implement within the timeframe of this study.

- Analyzing unit test coverage was not possible unfortunately as it would require a completely different tool that would perform dynamic analysis by compiling the given unit tests and producing the reports. Tools that implement analysis of unit tests, for example SourceMeter, are not free and on top of that, they do not do any compilation, only the analysis of the already generated reports.

4.7 Analyzed open source software

A set of open source systems that is analyzed in this study is as follows:

- Apache Batik SVG Toolkit - toolkit for applications or applets that want to use images in the Scalable Vector Graphics (SVG) format for various purposes, such as display, generation or manipulation. (The Apache Software Foundation, n.d.)
- Checkstyle - development tool to help programmers write Java code that adheres to a coding standard. It automates the process of checking Java code to spare humans of this boring (but important) task. This makes it ideal for projects that want to enforce a coding standard. (Anon., n.d.)
- Cobertura - free Java tool that calculates the percentage of code accessed by tests. It can be used to identify which parts of your Java program are lacking test coverage. (Cobertura, n.d.)
- DrawSWF - simple drawing program written in Java2. Drawings can be saved as animated flash file. (Anon., n.d.)
- FindBugs - program which uses static analysis to look for bugs in Java code. (Anon., n.d.)
- FreeCol - a turn-based strategy game based on the old game Colonization, and similar to Civilization. The objective of the game is to create an independent nation. (Anon., n.d.)
- FreeMind - free mind-mapping software written in Java. (Anon., n.d.)
- GeoTools - an open source Java library that provides tools for geospatial data. (Anon., n.d.)
- Heritrix - extensible, web-scale, archival-quality web crawler project. (Anon., n.d.)
- Informa - a news aggregation library based on the Java Platform. (Anon., n.d.)
- JAG - an application that creates complete, working J2EE applications. It is intended to alleviate much of the repetitive work involved in creating such applications, while providing a means of quality assurance that the applications created will be of consistent quality. (Anon., n.d.)
- JGraphT - Java graph library that provides mathematical graph-theory objects and algorithms. (Anon., n.d.)
- JHotDraw - a two-dimensional graphics framework for structured drawing editors that is written in Java. (Anon., n.d.)
- JTOpen - a library of Java classes supporting the client/server and internet programming models to a system running IBMi (or i5/OS or OS/400). The classes can be used by Java applets, servlets, and applications to easily access IBMi data and resources. (Anon., n.d.)
- Log4J - a logging library for Java. (Anon., n.d.)
- MegaMek - an unofficial, online version of the Classic BattleTech board game. (Anon., n.d.)
- NekoHTML - a simple HTML scanner and tag balancer that enables application programmers to parse HTML documents and access the information using standard XML interfaces. (Anon., n.d.)
- Quilt - a Java software development tool that measures coverage, the extent to which unit testing exercises the software under test. (Anon., n.d.)

- Apache Tomcat - an open source software implementation of the Java Servlet, JavaServer Pages, Java Expression Language and Java WebSocket technologies. (Anon., n.d.)
- Vuze – powerful Bittorent client. (Anon., n.d.)

CHAPTER 5: ANALYSIS RESULTS

In this chapter, the 20 open source systems that were described in the "Analyzed open source software" section will be examined. It is worth noting, that these systems weren't chosen randomly. The main criteria when selecting systems to be analyzed was their overall size hoping that it can later be used to discover trends between larger and smaller systems.

Table 5.1 shows the metric set against which the OSS systems were analyzed using combination of custom and third-party analysis tool mentioned in the "Analysis Tools" chapter.

Table 5.1 List of metrics that were analyzed

METRIC	DEFINITION
LOC	Total number of lines of code
STMTS	Total number of statements
CMTS	Percentage value of comments within code
CLS	Total number of classes
FNC	Total number of functions
AFC	Average number of functions per class
ASF	Average number of statements per function
ACC	Average complexity per class
ACF	Average complexity per function
IO	Number of short name variables (1-3 char.) with upper case "I" or "O" within name
SLVN	Number of variables with single letter names (excluding loop variables)
AVNL	Average variable name length
AFNL	Average function name length
ACNL	Average class name length
DUPL	Percentage of duplicated code blocks
POL	Percentage of polyadic functions
OVER20	Percentage of functions over 20 lines long (declaration, comments and whitespaces excluded)

Tables 5.2 and 5.3 show the numerical summary of the 17 metrics. It would be hard to draw any conclusions or observations without any graphical visualization. Of particular interest, however, is the number of short name variables with upper case "I" and "O" within their names and the number of single letter variable names. Before receiving these results, it was planned to show this number as a relative percentage of the entire variable count, and even though it was expected for that percentage to be really low, the absurdly low amount of the variables that met this criteria nearly across the board showed that this "Clean Code" characteristic is followed really well with a rare occasions of violations. In fact, single letter variable names are mainly present in algorithmic language and the projects that have had most of them were GeoTools and Megamek. Geotools is a java library with tools for geospatial data and Megamek is a game, both of which are expected to have large amount of algorithmic functions, therefore the occurrences of single variable names are within those systems. On top of that, there is also a direct correlation to size since only the larger projects had single letter variable names. It can be assumed as the system size increases, there is a higher chance that this characteristic will eventually be overlooked from time to time. While a violation of "Clean Code" characteristic, the impact is most likely nonexistent because of rare occurrence.

Table 5.2 Numerical summary of the results

System name	LoC	Stmts	Cls	Fnc	AFC	ASF	ACC	ACF
Batik	196,061	90,131	2,599	16,086	5.8	6.23	17.5	2.8
CheckStyle	72,563	24,250	2,026	5,970	3.09	3.94	6.4	2
Cobertura	19,024	8,479	270	1,454	5.48	5.87	11.8	2.1
DrawSWF	27,674	13,658	309	2,663	7.63	5.51	18.5	2.1
FindBugs	131,374	62,850	1,641	9,960	6.21	6.09	24.8	3.7
FreeCol	134,816	91,085	1,388	9,530	6.25	7.61	30.7	2.6
Freemind	67,298	30,298	907	6,542	6.14	4.88	15.7	2.2
GeoTools	1,013,328	452,527	11,404	92,053	7.6	4.71	18.5	2.3
Heritrix	47,655	21,107	596	4,371	6.27	5.15	16.7	2.3
Informa	28,879	6,279	219	1,130	6.97	4.26	10.5	2
JAG	15,699	8,585	136	1,091	5.37	7.16	24.7	3.1
JGraphT	26,202	11,675	398	1,860	4.34	6.48	10.9	2.3
JHotDraw	32,430	16,470	356	3,477	7.25	4.68	20.8	2.1
JTOpen	392,010	174,196	2,372	25,166	10.34	6.76	36.7	3.4
Log4J	35,217	16,522	478	3,321	6.11	5.22	15.3	2.2
Megamek	280,645	166,043	2,082	13,287	5.16	12.65	31.2	4.9
NekoHTML	7,972	3,887	68	542	6.73	7.16	30.6	3.8
Quilt	8,020	3,847	113	692	6.43	5.21	14	2.3
TomCat	280,645	135,317	3,137	23,761	7.62	5.57	21.2	2.8
Vuze	598,626	260,141	4,305	38,435	4.62	6.31	28.8	3.2

Table 5.3 Numerical summary of the results

System name	Cmts	IO	SLVN	AVNL	AFNL	ACNL	DUPL	POL	Over20
Batik	23.1	2	0	10.75	10.55	8.08	6.8	5.5	7.3
CheckStyle	32.8	1	0	10.51	12.73	8.2	13.2	2.0	5.2
Cobertura	14.8	0	0	8.82	13.44	6	5.4	6.0	7.6
DrawSWF	15.8	0	0	7.06	13.94	9.94	4.3	10.3	6.9
FindBugs	15	0	1	8.05	13.19	7.64	1.2	3.9	7.8
FreeCol	21.6	0	0	9.98	15.12	8.87	1.4	4.2	9.8
Freemind	15.5	1	0	8.55	13.62	9.28	2.1	3.8	5.3
GeoTools	32.7	4	15	7.01	11.9	7.42	10.5	3.8	6.9
Heritrix	24.3	7	0	10.94	16.3	7.93	1.8	3.3	5.8
Informa	36.9	0	1	7.47	16.23	5.98	6.7	3.5	6.2
JAG	19.3	0	0	10.28	13.76	7.34	1.5	1.6	15.2
JGraphT	27.7	0	0	9.29	14.52	7.96	4.5	3.3	10.2
JHotDraw	22.1	0	0	8.25	12.2	4.17	12.1	4.0	4.7
JTOpen	34	0	2	10.46	12.94	6.74	14.7	4.5	8.7
Log4J	26.5	1	0	8.43	12.22	7.19	8.5	4.8	5.2
Megamek	13.1	0	21	9.26	10.65	6.9	15.9	6.3	13.5
NekoHTML	27.3	0	0	10.72	10.33	5.91	1.1	7.4	8.9
Quilt	25.8	0	0	8.35	15.9	5.23	3.9	0.4	8.4
TomCat	21.5	0	3	7.54	14.83	5.1	5.1	3.5	7.4
Vuze	7.9	0	9	7.73	15.22	7.54	4.3	4.1	13.7

5.1 Names

Next we examine average variable, parameter and function name lengths as shown in figure 5.1. As expected, variable names require the least amount of characters to be expressive (between 7-10 characters) while there is a significant increase for class and function name lengths. Class names seem to be somewhere between 20-25 characters while function names appear to be somewhere between 27-35 characters. Due to the nature of functions as a unit within source code, they tend to be comprised of several words at least, therefore the length.

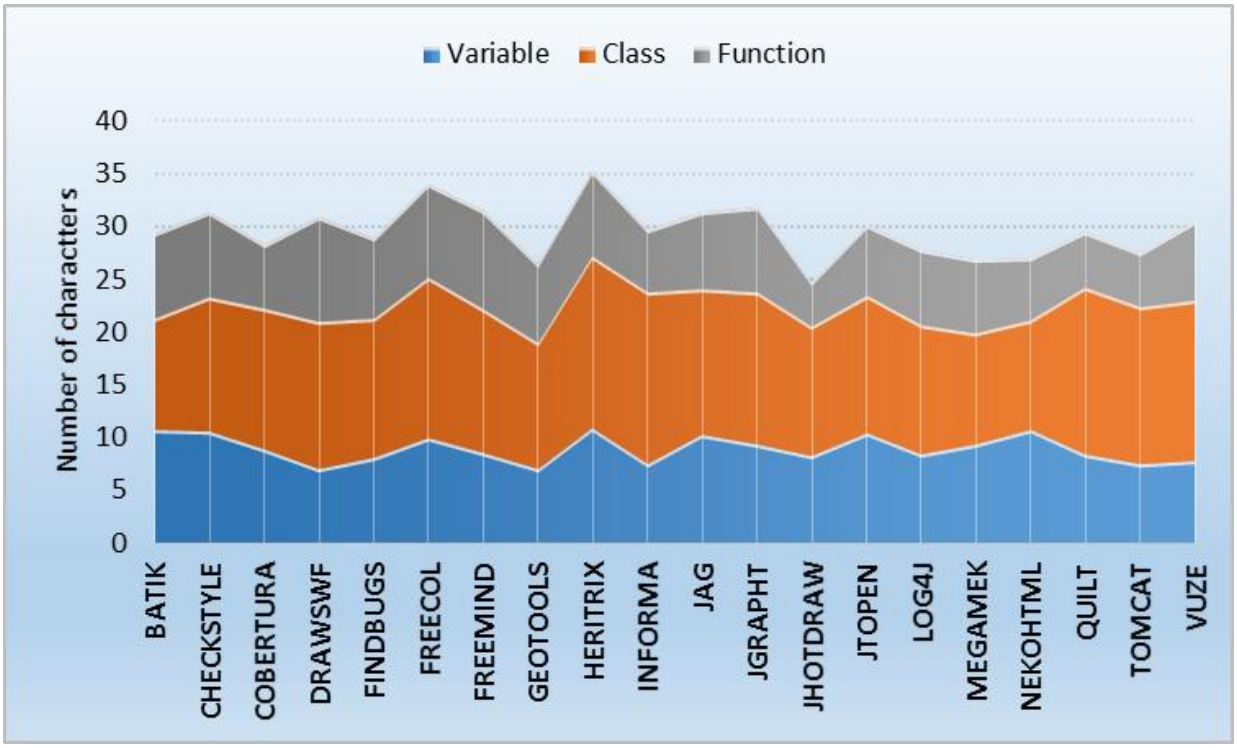


Figure 5.1 Average variable, class or function name length

Overall, it appears that there exists regularity amongst all 20 analyzed OSS systems, the difference being around 5 characters. However, notice the graphs themselves, especially the curving. A completely unexpected and interesting finding here is that it looks like all graphs are curved roughly the same way. If one system has longer variable names than the other, that relative difference seems to also hold for class or function names. Heritrix has the highest average name lengths, does that mean it has the most expressive names, therefore the cleanest code? Perhaps, but only partially. The real reason is actually not as obvious, it requires a little bit of manual code inspection of these systems. Nearly every software system has some domain specific language within the source code. Very often the same prefixes are used over and over when naming variables, classes or functions. The 20 systems that were analyzed were not exceptions to this. Depending on their domain specific language and the length of those affixes that were used when naming, it has offset their average lengths to some extent. For example, Heritrix has 18 classes with the word "*Extractor"* and 13 classes with the word "*Heritrix"* in their names, while Freemind has 29 classes with the word "*Map"*, thus the 4 character difference in average class name lengths between these two systems.

Finally, no apparent and meaningful deviations could be found when it comes to name lengths. Across the board the average name lengths are more than appropriate to be descriptive.

5.2 Size

The most well-known size metric for software is lines of code. As mentioned earlier, a variety of systems were chosen with 2 extremely large (500,000-1,000,000 LoC), 7 large (100,000-500,000 LoC), 5 average (30,000-100,000 LoC) and 6 small (5,000-30,000 LoC) systems as can be seen in figure 5.2.

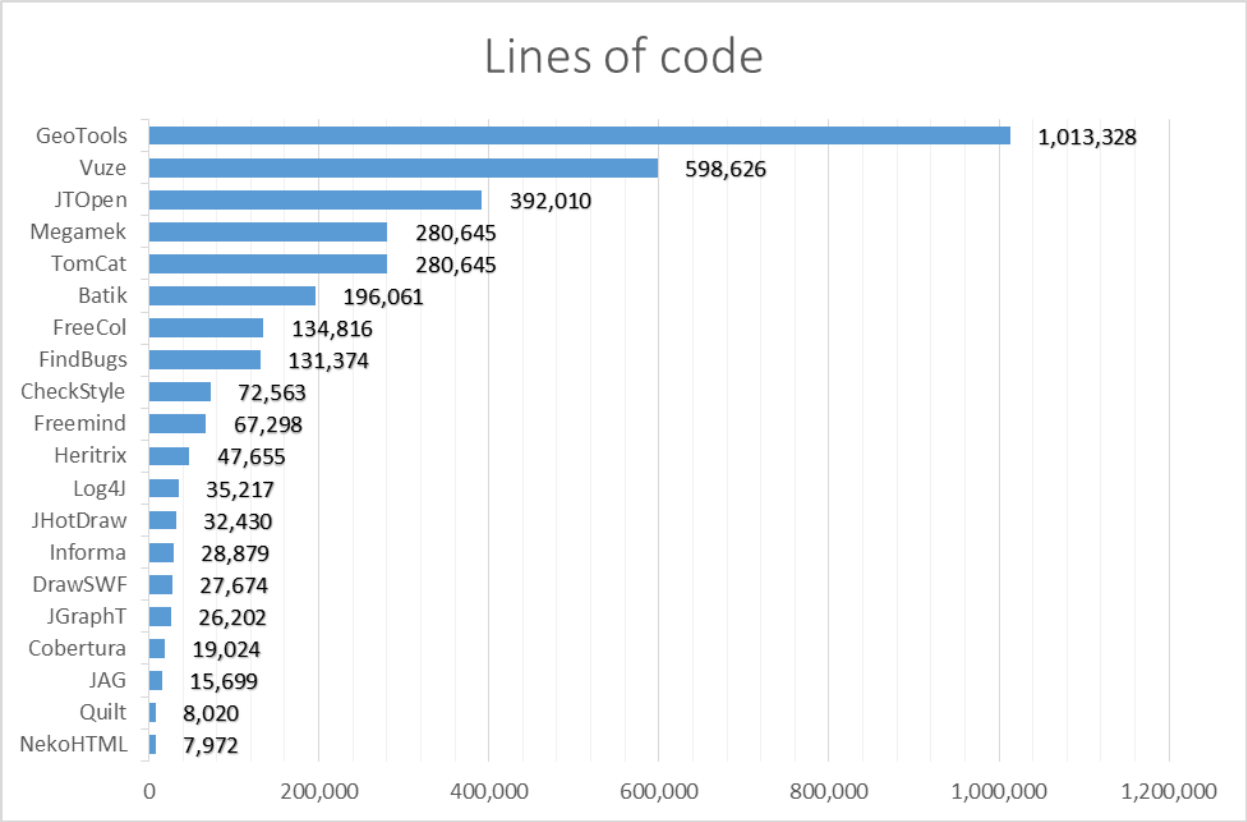


Figure 5.2 Total lines of code

5.2.1 Function size

Unfortunately, overall software size alone cannot really give any relevant indication as how to clean the system is. Figure 5.3 depicts a more appropriate metric - statements per function. The whiskers indicate the minimum and maximum number of statements in a single function of all the functions in a given system. The boxes represent the degree of distribution.

The reason why it was chosen to use statements per function and not lines of code per function as the first size metric is because there can technically be more than logical code statement written in a single line of code, thus potentially obscuring the results. In the book “The Economics of Software Quality” Capers Jones and Olivier Bonsignour even wrote that “it is astonishing that a field called ‘software engineering’ would use a metric with such a huge range of variations for more than 50 years without creating effective counting standards” (Jones & Bonsignour, 2011).

Results show that most systems are not exceeding more than 10 statements per function on average, with the exception of Megamek. According to Robert C. Martin functions should rarely exceed 20 lines, and 10 statements will never convert into 20 lines. With that said, this does not mean that every function examined was less than 10 statements. The vertical scale in figure 5.3 is logarithmic and as we can, besides FreeCol, FindBugs, Vuze and DrawSWF, there were systems with extremely large functions reaching 70-80 statements long. On top of that, most of these systems contain a heavy amount of public API functions that are one line long, slightly offsetting the true average. The true champion belt for this particular metric has to be shared by FreeCol and FindBugs. Both of these systems have showed that despite the size (134,816 LoC and 131,371 LoC) it is possible not to go overboard when writing functions. It is also extremely impressive how GeoTools managed to produce the entirety of codebase (1,013,328 LoC) while averaging around 5 statements in a single function.

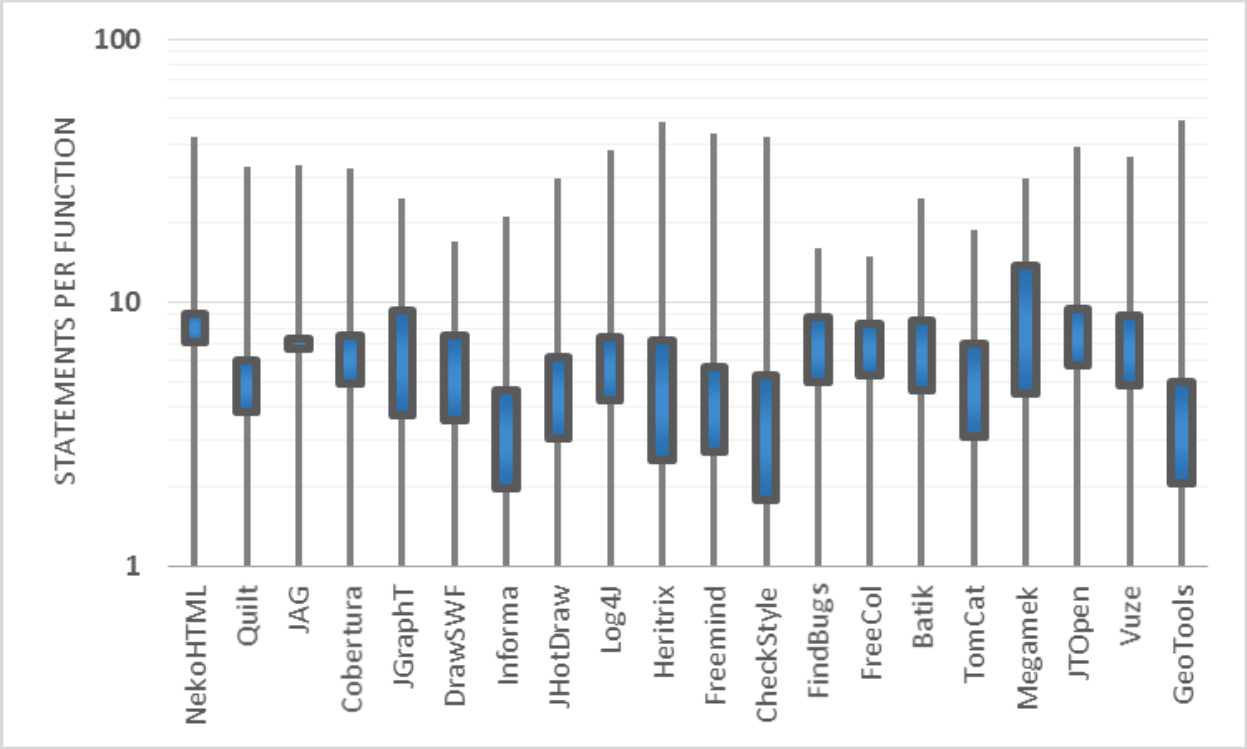


Figure 5.3 Statements per function distribution

Moreover, it can also be seen that there is quite a significant difference in the actual statement per function distribution in a single system. Some of the systems seem to have most of their functions at around the same number of statements, while others show a fairly huge variance in distribution. For example, Java Application Generator system has majority of its functions containing 7-8 statements while CheckStyle has 3-7 statements per function on average. It is quite odd finding and the only reasonable explanation I was able to find is that the smaller systems tend to have the smaller variance in the amount of statements per method used. This would require a more extensive investigation and probably a larger sample size in order to confirm or deny, but a speculation can also be made that generally smaller systems will have less amount (if not only one) of developers working on it, thus different preferences on the exact size of functions.

Figure 5.4 shows a different point of view – percentage of functions that were over 20 lines long. This time lines of code was used as a metric. It was established that longer functions tend to have several abstraction levels thus directly affecting readability. In the ideal world, the chart shown in figure 5.4 should have a single point in the middle, meaning that none of the systems have functions that are longer than 20 lines. In reality, however, the most prominent violators of this rule are Vuze, Java Application Generator and Megamek. These 3 systems have around 15% of their functions exceeding 20 lines. It was really surprising how such a small system like Java Application Generator (15,699 LoC) managed to clutter the codebase with so many long functions. In order to find the reason, manual code inspection was needed once again. The source of this evil was the *switch* statement – a fairly large chunk of functions were infested by it. They were simply bound to be long due to the nature of *switch* statement. It would be probably be a high chance to win with a bet that most of these functions from all systems that were over 20 lines long had some *switch* statement (this wasn't analyzed, however).

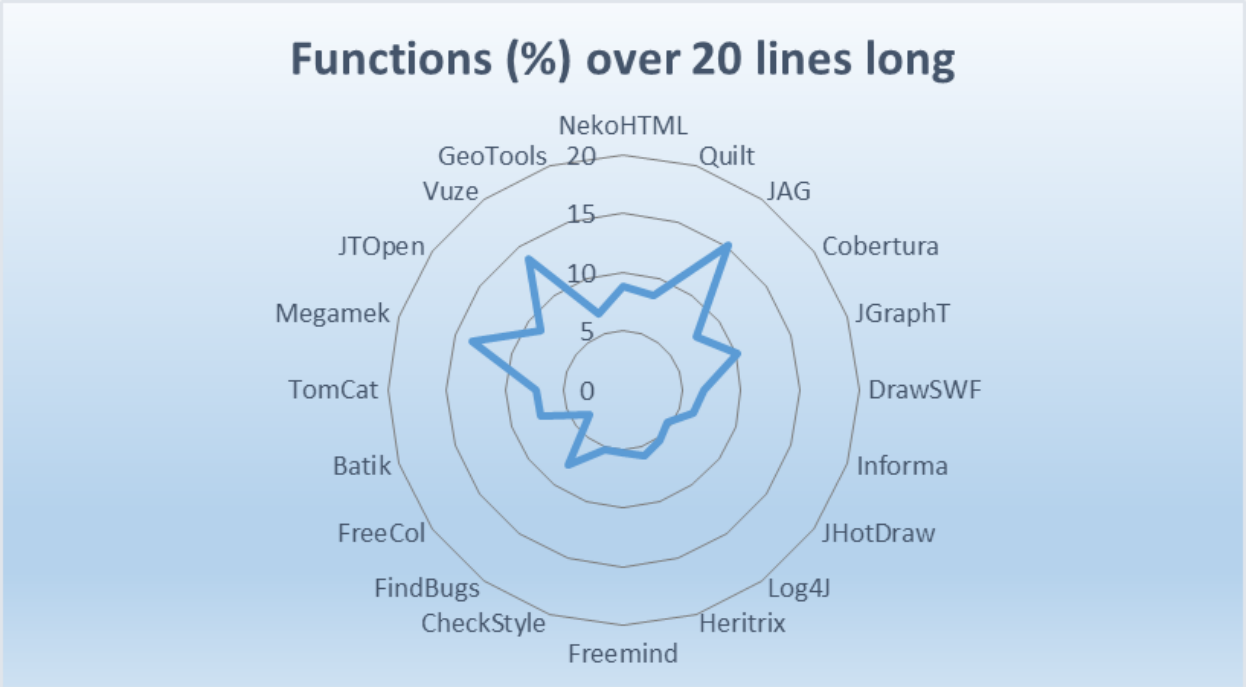


Figure 5.4 Percentage of functions that were over 20 lines long

5.2.2 Class size

After functions, next organizational level are classes. While function size can be measured by either lines of code or statements, class size is much harder to measure. The most appropriate metric would be amount of responsibilities, however, currently there are no effective methods discovered how to measure responsibilities themselves, therefore an assumption has to be made that classes with more functions are more likely to have more responsibilities. The exact ratios or thresholds are unclear.

Figure 5.5 shows the distribution of functions per class. It was designed with the same logic as figure 5.3 where whiskers indicate the minimum and maximum number of functions per class while the box represents degree of distribution. It is worth noting, that classes with no functions were excluded from the analysis as they can be considered data structures therefore 1 is the minimum. With the exception of JTOpen, all systems show the regularity of either staying below or barely exceeding 10 functions per class

on average. This number might once again be artificially increased due to fact that most of these systems have a large number of public API functions. It appears that the distribution of functions varies system by system. Informa, NekoHTML, Quilt, Heritrix, FreeCol and Cobertura show the biggest similarity of having their classes with the size of 5-10 functions on average. GeoTools, TomCat and JHotDraw seem to have written most of their classes with 8-11 functions on average. It can be assumed that the difference in deviation could probably be attributed mostly to the design of the system. Overall, this metric appears to be particularly vague, because it doesn't seem to be affected by anything or connected to any other metric.

Another interesting observation from figure 5.5 is the size of largest functions within those systems. Keep in mind, that once again, the vertical scale is logarithmic, therefore the increase is huge. It seems that none of the analyzed systems managed to get away without dumping some classes with extremely large amount of functions. It would certainly be a safe bet to say that those classes have more than one responsibility. The reigning champion is Megamek with 808 functions in one class called "Entity.java" (now that is an accomplishment). In fact, that class was 13,385 lines long. The authors of source code have described the class as "A master class for basically anything on the board except terrain" and it pretty much contained variables and objects about pretty much every other aspect of the system. Even if there were no variables in that class, it would average out about 16 lines per function, which is actually not that bad. The runner-up is GeoTools with class containing 408 functions. It would be hard to find the exact reason for the existence of classes with such a huge amount of functions without actually speaking to the designers of the system. Theoretically, it could be connected to size, acting as super classes for these large systems, however, it might very well be the laziness of the developers to properly decouple these classes into smaller components.

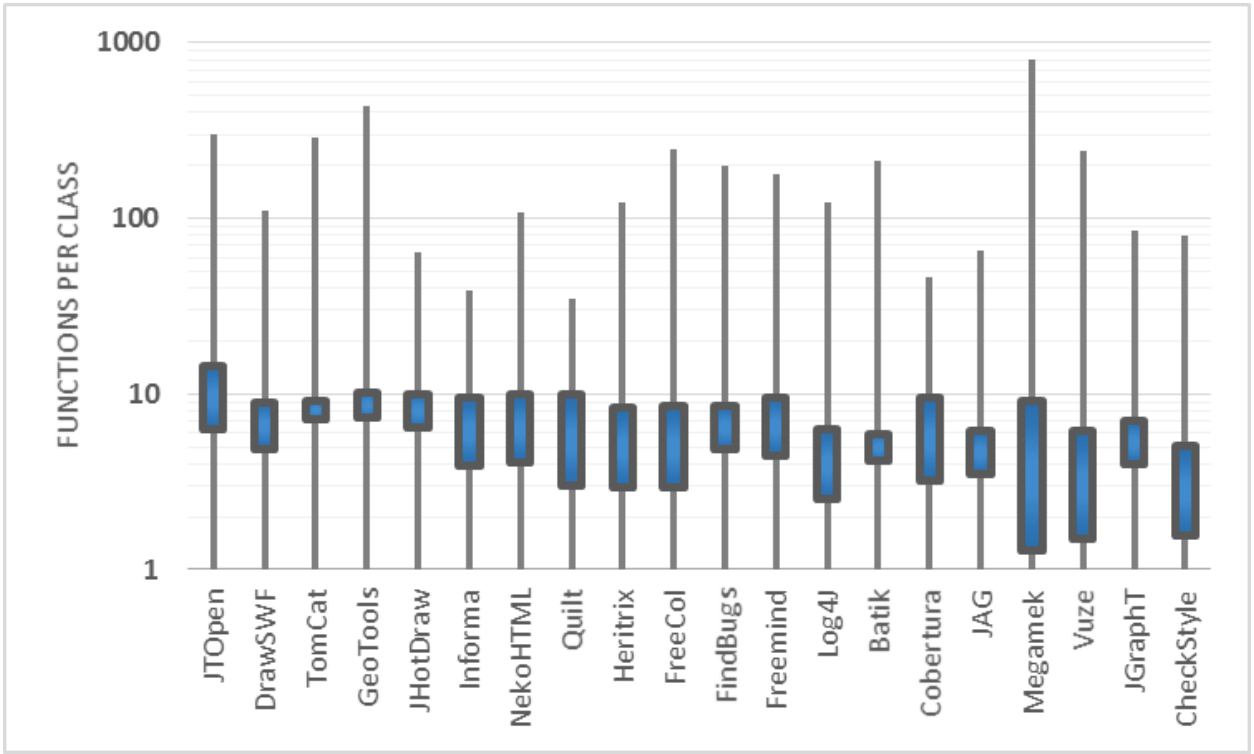


Figure 5.5 Function per class distribution

5.2.3 Alternative look

So far, the systems have been examined separately, but what about the big picture? Figure 5.6 shows the distribution of number of statements per function of all the systems. Vertical lines indicate the highest and the lowest percentage of statements per function that was found in one of the systems, while the circular mark shows average percentage value. For example, the lowest percentage of functions with 2 statements in one of the systems (system name is not indicated on the chart) was 3.8%, the highest (system name is also not indicated) – 6.8% and the average in 20 systems – 6.6%.

That's actually a fairly large sample size – over 1.5 million functions. The regularity can be seen from around 4-8 statements per function. This is really impressive since each of those sizes seem to represent about 10% of all the functions analyzed, which makes it around 50% total. Another 20% is 1-3 statements per function. The slight drop after one statement per function to two happens because there are generally a lot of types of functions, such as getters, setters, utility and most public API functions that usually have only one statement. Gradual decrease occurs in 9-17 statements per function range covering about 20% of the total. Finally, the rest of the range drops below 1% total for each size and covers a mere less than 10% of the 1.5 million functions analyzed.

In general, there is an obvious preference towards shorter functions, which means that developers of these systems understand the importance of readability and do not want to make life hard for the readers of the code as well as themselves. Moreover, these results are consistent with the ones in figure 5.6 where the percentage of functions with over 20 lines was shown. While 20 lines will not always directly convert into 20 statements, we can see that around 10% of all functions had more than 20 statements and it would easily present the values in figure 5.6.

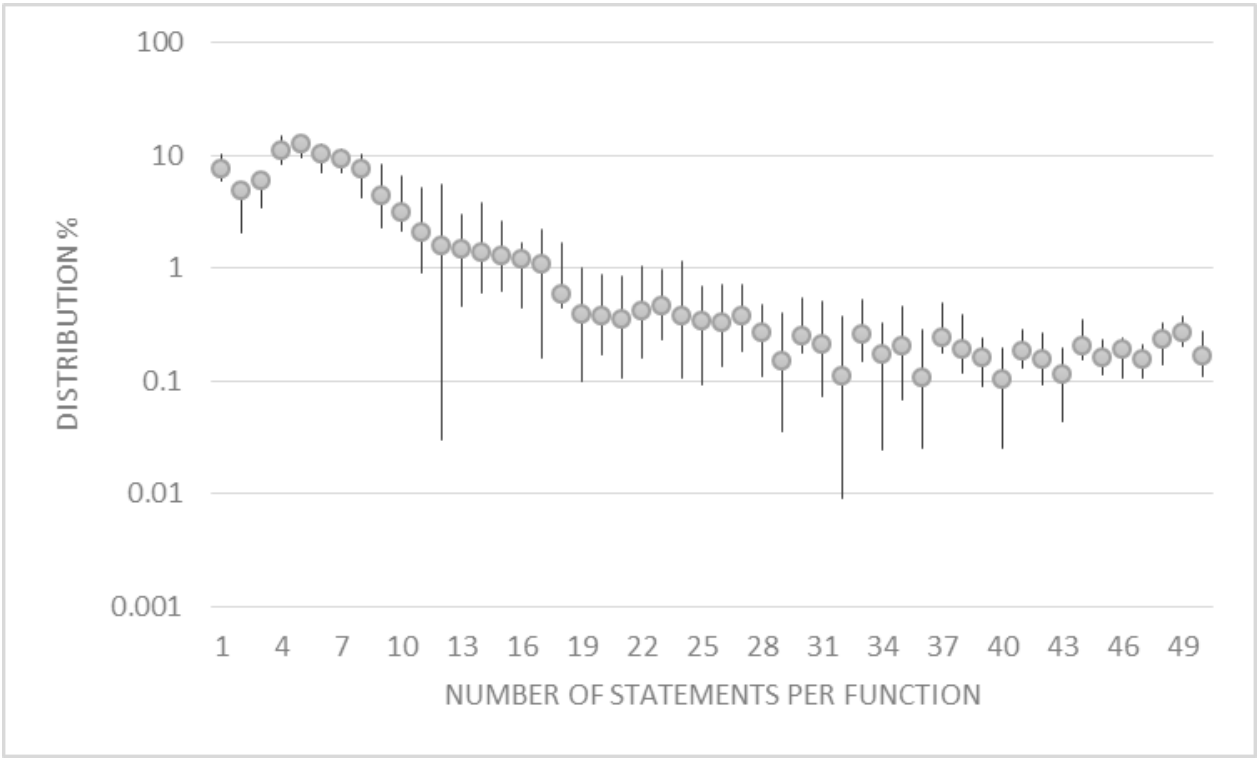


Figure 5.6 Overall statement per function distribution

5.3 Function parameters

In the book "Clean Code: A Handbook of Agile Software Craftsmanship" Robert C. Martin says that:

> The ideal number of arguments for a function is zero (niladic). Next comes one (monadic), followed closely by two (dyadic). Three arguments (triadic) should be avoided where possible. More than three (polyadic) requires very special justification—and then shouldn't be used anyway (Martin, 2009).

Figure 5.7 showcases the percentage of functions that "shouldn't be used" in the analyzed OSS systems. The eye should instantly be drawn to that nearly nonexistent bar for Quilt. Even though it is one smallest systems in the analyzed set, the developers should definitely be congratulated for being able to write those 8,020 lines of code while only producing around 0.4% of all functions that had more than 3 parameters. On the other hand, the biggest violator of this characteristic is DrawSWF, where over 10% of functions were polyadic. The rest average somewhere between 3-6%. Actually the only two systems that deviated from average were DrawSWF and NekoHTML. The reason for this deviation is different for each of these systems: the majority of the polyadic functions in DrawSWF were multiple constructor overloads while NekoHTML had around 60% of those polyadic functions with 4 parameters. It is entirely likely that the latter reason is a valid excuse if there is a justification, such as parameters not belonging to one common object. As for the DrawSWF, perhaps the analysis should have actually omitted constructor functions since the easiest way to reduce the number of parameters within functions is to form objects, which is exactly what constructor functions are used for.

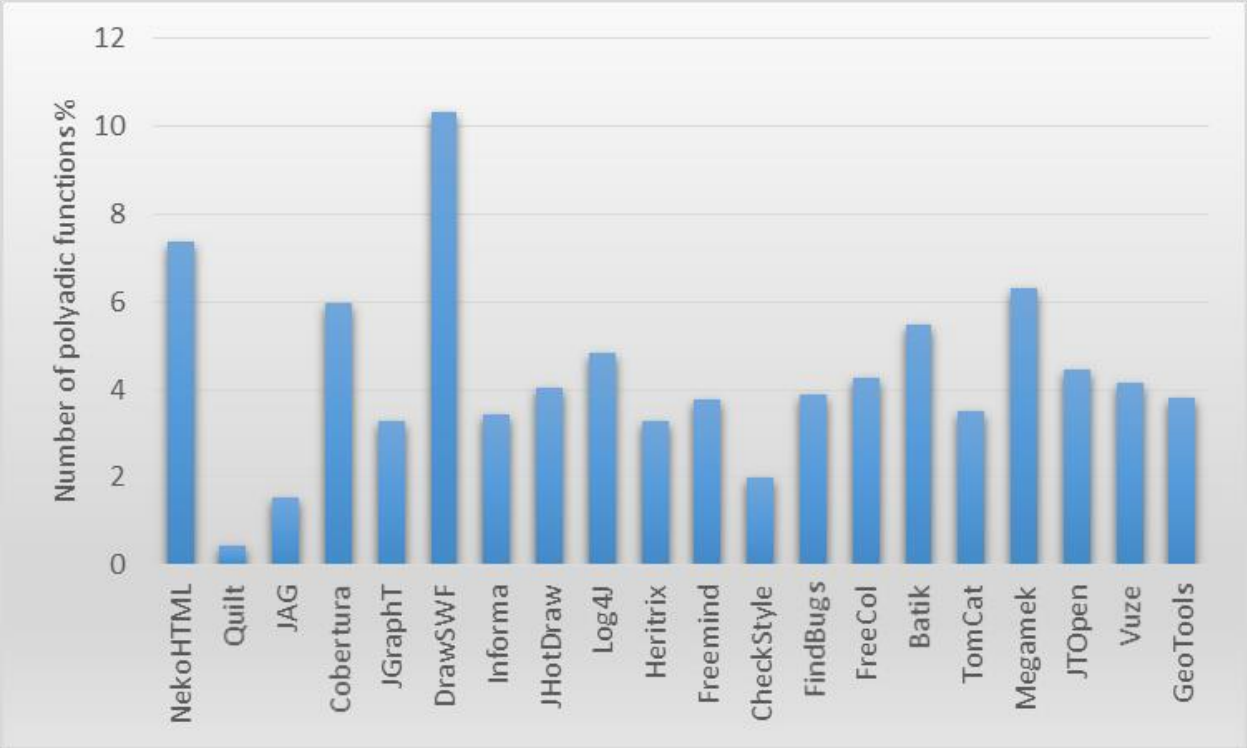


Figure 5.7 Percentage of polyadic functions

The overall analysis of over 1.5 million functions of all systems shown in figure 5.8 once again yields some impressive results. The code is reasonably clean with only 4.52% of the functions being polyadic. Astonishing and unexpected, however, is the domination of the niladic functions. Expectation was for the monadic and dyadic functions to take the first and second places. To be honest, the fact that functions with zero, one or two arguments take around 85% of the total pool showcases absurdly high friendliness to unit tests and how the developers were able to decompose problems into smaller and individually simpler tasks.

These findings can possibly be related to the statements per function distribution chart in figure 5.3. It is no surprise that functions with less parameters are likely to perform less operations and thus result in fewer statements. The logic is consistent within the two charts: around 85% of total functions contained 1-13 statements which could cover functions with few parameters. The rest had more and the 15% should relate to triadic and polyadic functions. This is, however, only a hypothesis and without more in depth analysis shouldn't be taken for granted.

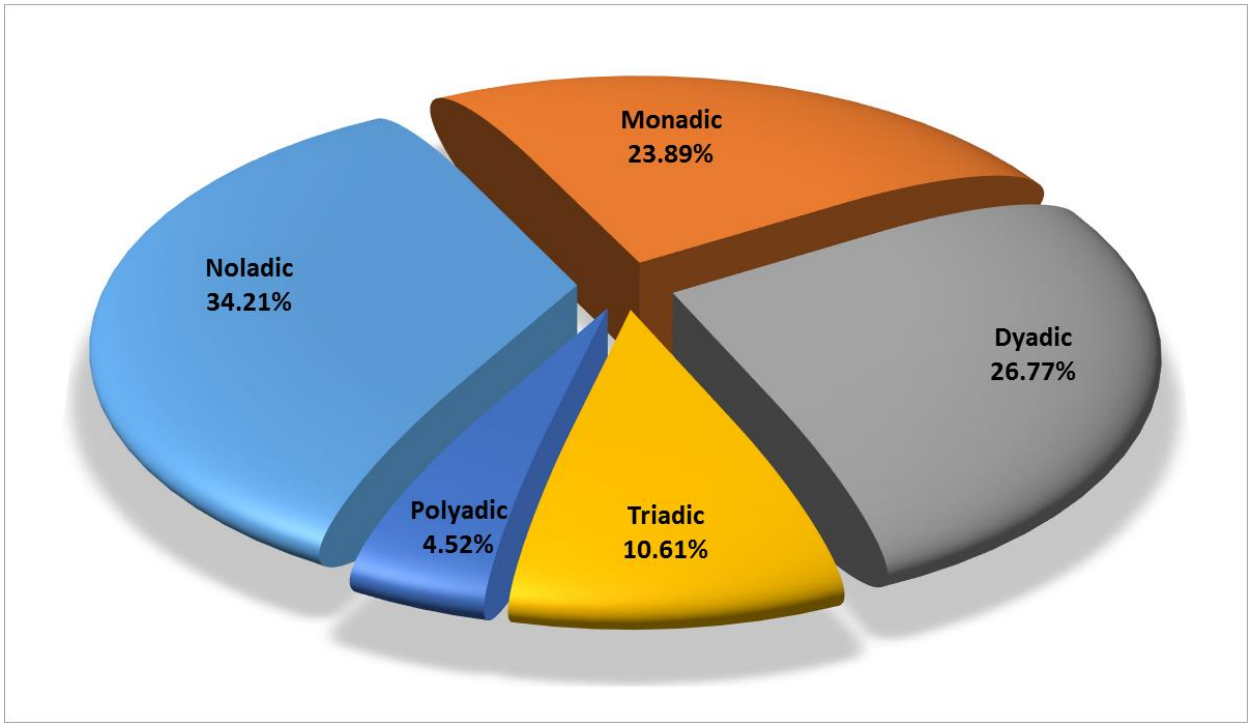


Figure 5.8 Overall parameter per function distribution

5.4 Comments

The treemap in figure 5.9 depicts the usage of comments within the OSS systems. The size of one item corresponds to the lines of code and the color corresponds how well commented the system was (blue-high to white-low spectrum). It is worth noting, that legal comments were omitted from this analysis.

The results do not really show any trends, which is fairly unexpected. The usefulness of comments would require analyzing the semantics and is outside of the scope of this study, but it appears that despite the fact that 'Clean Code' guidelines suggest to refrain from obscuring the code with a lot of documentation

comments, the majority of the systems are still using them excessively. With that said, the general expectation would be for the smaller systems to be commented the most because it might be assumed that, as the project grows larger there might be less inclination to comment, especially as changes are made. However, it is possible for such assumption to be completely faulty because the data is conflicting and contradicts it. Projects like GeoTools and JTOpen appear to have managed to maintain their source code commenting regardless of size. On the other hand, Vuze is also a large system but severely lags behind. Perhaps the sample size of 20 OSS systems might be too small in order to uncover more meaningful trends.

A middle ground solution that would satisfy both, 'Clean Code' principles for commenting and the systems' need to have documentation, could be an automatic documentation generator tool, allowing to separate documentation from the actual source code. The tool would inspect source code without documentation comments and produce them automatically without cluttering the original source. Output comments could then be parsed by another tool in order to create documentation. Some researchers have already been fairly successful in accomplishing such task – see the related work section.

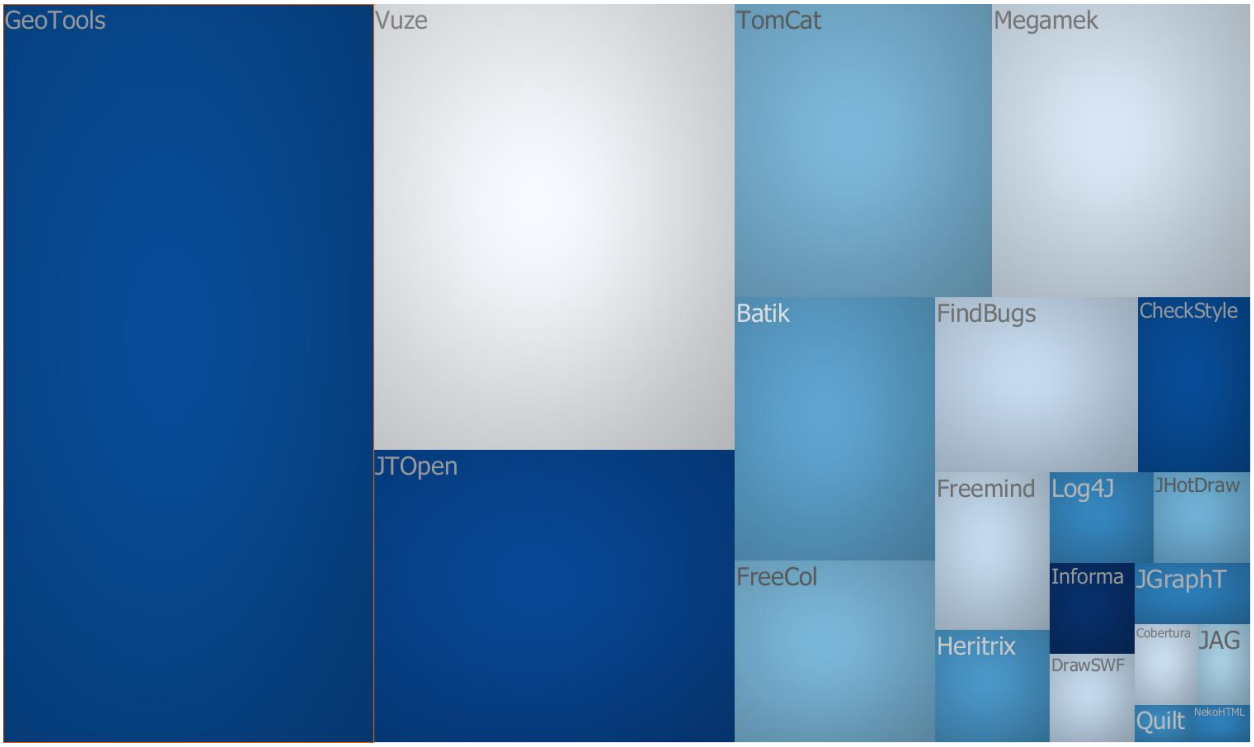


Figure 5.9 System size (square size) and percentage of comments (color) comparison

5.5 Complexity

Probably the vaguest of the metrics that were used for analysis in this study is complexity. How exactly does it affect cleanliness of the code? Well, it shouldn't come as a surprise that the more complex code is, the harder it is to read it, thus also harder to enhance it. Average complexity per class and function is shown in figure 5.11. Of particular interest here, are the Java Application Generator, Megamek and Vuze. Seemingly high average complexity for these systems relate directly back to the findings from functions that were over 20 lines long chart in figure 5.4 where it was identified that extensive usage of *switch* statement has caused abnormal amount of functions to be larger than expected. The same reason holds for complexity - *switch* statement appears to be the highest contributor to this metric.

Moreover, in order to see another trend, we have to introduce an additional metric – average statements per class. The numerical summary for this metric is shown in figure 5.10. With the exception of Vuze, notice how close the order of the results are to the ones shown in figure 5.11. CheckStyle, Informa and JGraphT have low amount of statements per class and low average complexity per class while Megamek, FreeCol, NekoHtml and JTOpen have high average amount of statements per class and high average complexity per class. The conclusion can be drawn that classes and functions with more statements are simply bound to be more complex in most cases. Actually, it is quite natural that more intricate things require more information to explain them.

System name	ASPC
CheckStyle	12.2
JGraphT	28.1
Vuze	29.2
Informa	29.7
Freemind	30.0
Log4J	31.9
Cobertura	32.2
Heritrix	32.3
Quilt	33.5
JHotDraw	33.9
GeoTools	35.8
Batik	36.1
FindBugs	37.8
JAG	38.4
DrawSWF	42.0
TomCat	42.4
FreeCol	47.6
NekoHTML	48.2
Megamek	65.3
JTOpen	69.9

Figure 5.10. Average amount of statements per class

Another odd thing observed was that an average class and function complexity metrics don't seem to be linked. Intuitive prediction would be for the systems with more complex classes to also have more complex functions, however, this is not the case. The explanation lies in the density of statements in functions. Systems that tend to have their classes with a lot of small functions will result in having low average function complexity, but not necessarily low average class complexity. For example, a system with 1000 small functions that are contained in 100 classes will have different ratio for average complexity per class/average complexity per function than a system with 500 larger functions that are contained in 100 classes. Moreover, rearranging larger functions into simpler, smaller ones would not affect overall class complexity, but would affect average function complexity in that class, granted that the functions remain the same. The only way to affect class complexity is to either remove duplication by abstracting out duplicated blocks of code into separate functions or create more classes. However, note that the overall difference for average function complexity across 20 systems is close to being negligible. There is a clear preference towards writing functions that do not exceed complexity value of 4.

Of the 20 systems analyzed, Checkstyle has the lowest average class complexity while also having one of the lowest average function complexity. The highest value for these metrics goes to Megamek.

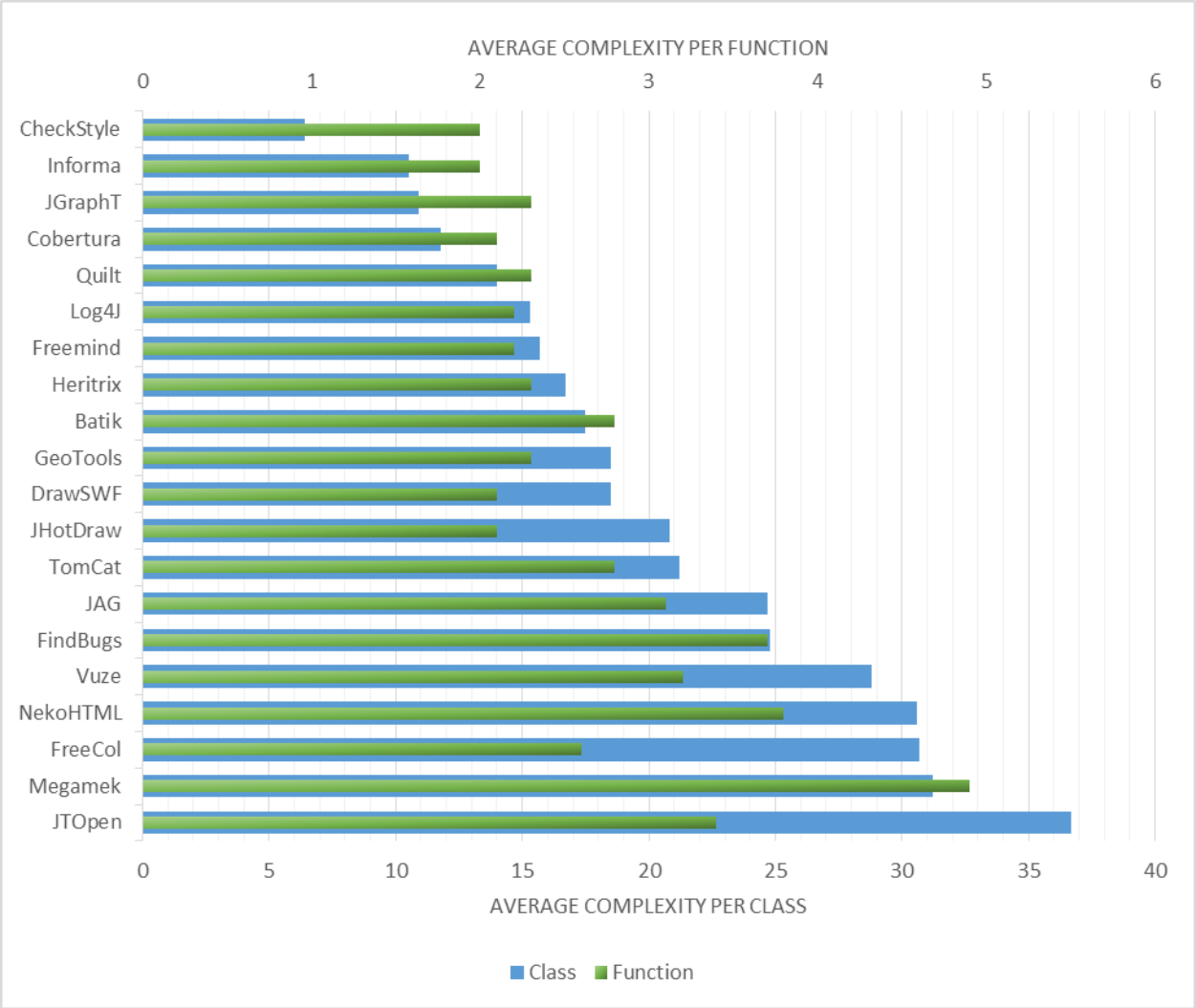


Figure 5.11 Average complexity per class and function comparison

5.6 Duplication

Duplication is an extremely relevant metric in software development and a sizable contributor to cleanliness of code. Figure 5.12 shows the percentage of code that is duplicated within the analyzed OSS systems. It appears to be related to the lines of code metric shown in figure 5.2. With the exception to JHotDraw, the top 5 are large systems, while the rest also follows the bigger-more duplication trend to certain extent. To be fair, it should be expected for the software size to be a factor in code duplication, since duplication is much more likely occur when there is lots of code around that a developer can just copy and paste instead of abstracting it into a new function or class. On top of that, the larger the project is, it also usually has more developers working on it. “Enhancing someone else’s code is always harder since first the understanding has to be developed and it is common to simply duplicate existing code” (Yarmish & Kopec, 2007). Finally, the OSS systems might have had random contributors that were not particularly motivated to come up with a more object oriented solution. However, last observation is a pure speculation with no data supporting it. Also it is not out of the realm of possibilities that a different kind of analysis with closed source systems would yield conflicting results.

If we were to pick a winner for the least duplication and system size ratio, it would definitely be FindBugs. Really impressive how 131,374 were written while only duplicating 1.2% of the source code. In the systems lifetime, FindBugs had 12 contributors to the source code, but the development team happily considers including feature enhancement from anyone willing to contribute. Another notable system would be Vuze where over half a million lines of code resulted in only 4.3% code duplication. All in all, the bottom 12 systems seem to have really put effort into reducing duplication while others not as much.

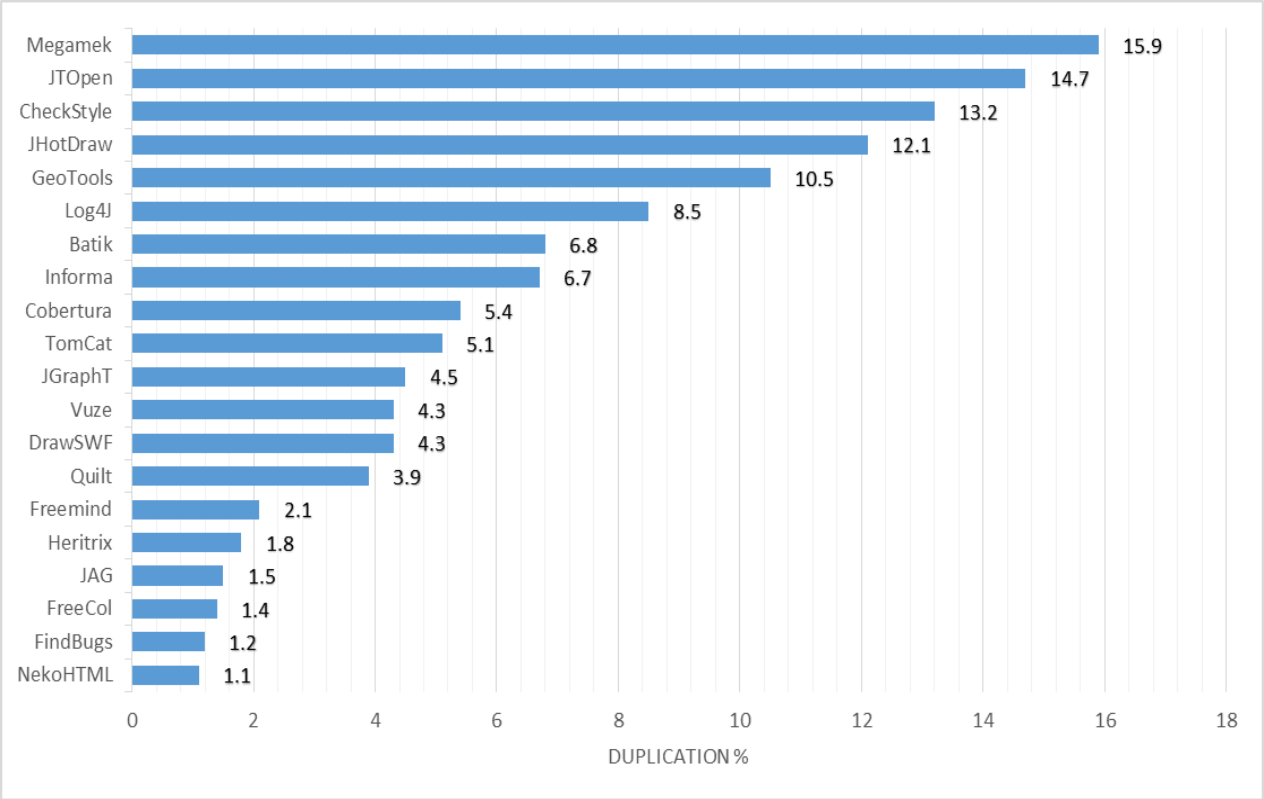


Figure 5.12 Percentage of duplicated code blocks comparison

5.7 Threats to validity

The main concern in terms of construct validity of this study is the fact the 17 code metrics that were measured are by no means a complete set in order to determine the extent of cleanliness of source code within software projects. There is an obvious lack in measuring important 'Clean Code' guidelines such as error handling and usage of unit tests, therefore the degree to which analysis could measure overall cleanliness of software system was limited.

Moreover, the selection of systems to be analyzed could also pose threat since open source systems tend to have a variety of developers contributing to the same project. The contributors may come from different backgrounds and have different emphasis on certain guidelines and principles. However, a comparable study by Diomidis Spinellis of the FreeBSD, Linux, Solaris and Windows operating systems showed that no obvious difference in overall quality could be identified between open or closed-source system types (Spinnelis, 2008).

In regards to external validity (extent to which the results can be applied to other systems), the sample size of 20 open source systems is relatively small and therefore a study covering the analysis of more systems could pose a threat to the observations, trends and relations between code metrics identified in this study.

Another threat to external validity is that generalizability of the findings could be impacted in a negative way due to the fact that the systems were selected based only on size in order to be able to use raw size as a metric. The type of the system (database, tool, editor, middleware, framework, etc.) was not taken into account. Metric like amount of comments within source code is definitely related to the system type as libraries or frameworks tend to have more comments due to larger amount of public API functions that are documented.

Finally, the analysis tools used for this study could possibly contain errors. SourceMonitor is less likely to have accuracy issues since it has had multiple revisions, but the custom tool developed for this study definitely might. Number of classes metric was cross measured for accuracy as it is also usually stated where the project source code is hosted, but other metrics could potentially have faulty measurements.

CHAPTER 6: CONCLUSIONS AND FUTURE WORK

This study describes the data gathered by analyzing 20 open source systems for selected source code metrics that are believed to have an impact to cleanliness of code. Based on this empirical analysis the following conclusions have been drawn:

- Commenting is the only characteristic (of the ones analyzed) that was not following the minimalistic principle of only writing comments when appropriate. Selected open source systems were cluttered with useless public API documentation that actually reduces readability.
- Short name variables are barely used in software projects nowadays and are only seen in highly algorithmic functions. Instead, longer, more descriptive names are preferred across the board.
- There is a direct relation between software size and other metrics, such as average number of function parameters, average complexity and the level of duplication.
- Only a small fraction of analyzed open source code contained functions that were more than 20 lines long and had more than 3 parameters showing a clear preference towards shorter functions with few parameters.
- In general, all the open source systems were following guidelines and principles (except for commenting) that make source code more readable and understandable to a decent degree.
- There was a clear difference in emphasis on certain 'Clean Code' characteristics. Some systems have produced better results in terms of one characteristic, but not necessarily the others.

Firstly, it is obvious that the usage of code metric tools can help evaluating cleanliness of source code. But can automatic code inspection possibly evaluate all 'Clean Code' characteristics? Unfortunately, it doesn't seem to be the case at the moment. More than often, in order to evaluate whether higher level source code units, such as classes or modules, follow key principles (Single Responsibility principle and Open-Closed principle), manual code inspection is required. Even though lower level characteristics such as naming or the usefulness of comments, would require analyzing the semantics of text. Even though some researchers are starting propose mechanical approaches, such as Ostwold and Host with their investigation of automatic check to determine whether the function's name and implementation is a good match (Host & Ostvold, n.d.), no concrete tools that can delve into semantics of source code have been developed yet.

Secondly, it is intriguing that there was no dominant 'Clean Code' characteristic uniformly followed by every system. In fact, the emphasis seemed to differ and material on the topic is also not consistent. The authors appear to prioritize some principles over the others. But is having an appropriate name for a function more important than the function following single responsibility principle? If so, then how much more important? Perhaps there is no need to prioritize at all? Further research is required in order to determine if it would even be possible to have different value on 'Clean Code' characteristics.

Lastly, the results showcased that it is possible to produce large code bases without severe violations of 'Clean Code' characteristics and principles. However, is there some additional burden for large systems to maintain cleanliness? Does the development methodology play the role? It is possible that a sequential design process with a clear vision of final product might be superior to incremental approach (such as Agile) when it comes to maintainability. Studying 'Clean Code' characteristics in large systems only might yield some interesting results on the relation between cleanliness and maintainability.

Areas of potential future research raised by this study are as follows:

- Analyzing the semantics of source code units (such as comments and function names) in order to determine the relevance to the associated implementation.
- Determining the relative importance of ‘Clean Code’ characteristics to the overall cleanliness of the code.
- Studying incremental growth of large software systems in order to find out whether guidelines and principles are followed throughout the development lifecycle and to what extent.
- Analyzing closed source systems could potentially yield different results.
- Studying systems selected by type (database, framework, tool, library, etc.) in order to uncover trends based on the system type.
- Analyzing cleanliness in concurrent code.
- Analyzing different aspects of ‘Clean Code’, such as error handling or unit tests.

REFERENCES

Anon., n.d. *Apache Tomcat.* [Online]
Available at: http://tomcat.apache.org/
[Accessed 10 July 2015].

Anon., n.d. *Checkstyle.* [Online]
Available at: http://checkstyle.sourceforge.net/
[Accessed 8 July 2015].

Anon., n.d. *Draw SWF.* [Online]
Available at: http://drawswf.sourceforge.net/
[Accessed 8 July 2015].

Anon., n.d. *FindBugs.* [Online]
Available at: http://findbugs.sourceforge.net/
[Accessed 9 July 2015].

Anon., n.d. *FreeCol - the Colonization of America.* [Online]
Available at: http://www.freecol.org/
[Accessed 9 July 2015].

Anon., n.d. *FreeMind.* [Online]
Available at: http://freemind.sourceforge.net/wiki/index.php/Main_Page
[Accessed 8 July 2015].

Anon., n.d. *GeoTools.* [Online]
Available at: http://geotools.org/
[Accessed 8 July 2015].

Anon., n.d. *GMetrics.* [Online]
Available at: http://gmetrics.sourceforge.net/index.html
[Accessed 26 July 2015].

Anon., n.d. *Heritrix.* [Online]
Available at: https://webarchive.jira.com/wiki/display/Heritrix/Heritrix
[Accessed 8 July 2015].

Anon., n.d. *Informa.* [Online]
Available at: http://informa.sourceforge.net/
[Accessed 9 July 2015].

Anon., n.d. *JAG - Java Application Generator.* [Online]
Available at: http://jag.sourceforge.net/
[Accessed 9 July 2015].

Anon., n.d. *JGraphT.* [Online]
Available at: http://jgrapht.org/
[Accessed 10 July 2015].

Anon., n.d. *JHotDraw.* [Online]
Available at: http://sourceforge.net/projects/jhotdraw/
[Accessed 9 July 2015].

Anon., n.d. *JTOpen.* [Online]
Available at: http://jt400.sourceforge.net/team.html
[Accessed 9 July 2015].

Anon., n.d. *Log4j.* [Online]
Available at: http://logging.apache.org/log4j/1.2/
[Accessed 10 July 2015].

Anon., n.d. *MegaMek.* [Online]
Available at: http://megamek.info
[Accessed 9 July 2015].

Anon., n.d. *NekoHTML.* [Online]
Available at: http://nekohtml.sourceforge.net/
[Accessed 10 July 2015].

Anon., n.d. *Quilt.* [Online]
Available at: http://quilt.sourceforge.net/
[Accessed 9 July 2015].

Anon., n.d. *Vuze.* [Online]
Available at: http://www.vuze.com/
[Accessed 10 July 2015].

Arsenovski, D., 2009. *Proffesional Refactoring in C# & ASP.NET.* Indianapolis: s.n.

Baldwin, C. & Clark, K., 2005. *Does Code Architecture Mitigate Free Riding in the Open Source Development Model?,* s.l.: s.n.

Buse, R. P. L. & Weimer, W. R., 2010. Learning a Metric for Code Readability. *IEEE Transactions on Software Engineering,* 36(4).

Campwood, n.d. *SourceMonitor.* [Online]
Available at: http://www.campwoodsw.com/sourcemonitor.html
[Accessed 29 July 2015].

Cass, S., 2015. *The 2015 Top Ten Programming Languages,* s.l.: IEEE Spectrum.

Cobertura, n.d. [Online]
Available at: http://cobertura.github.io/cobertura/
[Accessed 8 July 2015].

Collar, E. & Valerdi Ricardo, 2012. Role of Software Readability on Software Development Cost.

Coplien, J. O. & Bjornvig, G., 2010. *Lean Software Architecture for Agile Software Development.* s.l.:s.n.

Fowler, M. et al., 199. *Refactoring: Improving the Design of Existing Code.* s.l.:s.n.

Hassan, A. E., Mockus, A., Holt, R. C. & Johnson, P. M., 2010. Guest Editors' Introduction: Special Issue on Mining Software Repositories. *IEEE Transaction on Software Engineering,* 31(6).

Heusser, M., 2013. *CIO.* [Online]
Available at: http://www.cio.com/article/2386909/agile-development/how-to-deal-with-software-development-schedule-pressure.html
[Accessed 8 August 2015].

Host, E. W. & Ostvold, B. M., n.d. *Debugging method names.* s.l.:s.n.

Jones, C. & Bonsignour, O., 2011. *The Economics of Software Quality.* Massachusetts: Paul Boger.

Kuhn, A., Ducasse, S. & Girba, T., 2007. Semantic clustering: Identifying topics in source code. *Information and Software Technology,* 49(3), pp. 230-243.

Kuhn, T., 2006. *Abstract Syntax Tree.* [Online]
Available at: http://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html
[Accessed 16 April 2015].

Martin, R. C., 2002. *Agile Principles, Patterns, and Practices in C#.* s.l.:s.n.

Martin, R. C., 2009. *Clean Code: A Handbook of Agile Software Craftmanship.* s.l.:s.n.

McBurney, P. W. & McMillan, C., 2014. *ICPC, 22nd International Conference on Program Comprehension.* s.l., ACM.

McConnell, S., 2004. *Code Complete 2nd Edition.* s.l.:s.n.

Sinha, V., 2011. *Making Code Easy to Understand – What Developers Want (a study).* [Online]
Available at: http://blog.architexa.com/2011/07/making-code-easy-to-understand-what-developers-want-a-study/
[Accessed 23 June 2015].

Spinnelis, D., 2008. *A table of four kernels.* New York, s.n.

Sridhara, G., Pollock, L. & Vijay-Shanker, K., 2011. *Software Engineering (ICSE), 33rd International Conference.* Honolulu, IEE.

Stroustrup, B., 2008. *Programming: Principles and Practice Using C++.* s.l.:s.n.

Stroustrup, B., Dechev, D. & Pirkelbaur, P., 2010. *Source Code Rejuvenation Is Not Refactoring.* s.l., s.n.

The Apache Software Foundation, n.d. *Apache Batik SVG Toolkit.* [Online]
Available at: http://xmlgraphics.apache.org/batik/
[Accessed 9 July 2015].

Thomas, S. W., 2001. *Mining Software Repositories Using Topic Models.* s.l., s.n.

Wikipedia, 2015. *Single responsibility principle.* [Online]
Available at: https://en.wikipedia.org/wiki/Single_responsibility_principle
[Accessed 1 July 2015].

WikiPedia, n.d. *Model-View-Controller.* [Online]
Available at: https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller
[Accessed 20 July 2015].

WikiPedia, n.d. *Open/closed principle.* [Online]
Available at: https://en.wikipedia.org/wiki/Open/closed_principle
[Accessed 19 July 2015].

WikiPedia, n.d. *Separation of Concerns.* [Online]
Available at: https://en.wikipedia.org/wiki/Separation_of_concerns
[Accessed 20 July 2015].

WikiPedia, n.d. *SOLID principles.* [Online]
Available at: https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)
[Accessed 28 July 2015].

WikiPedia, n.d. *Template Method Pattern.* [Online]
Available at: https://en.wikipedia.org/wiki/Template_method_pattern
[Accessed 2015 July 25].

Yarmish, G. & Kopec, D., 2007. Revisiting novice programmer errors. *ACM SIGCSE Bulletin,* 39(2), pp. 131-137.