Bachelor Degree Project

Comparing functional to imperative Java

- with regards to readability, complexity and verbosity

Author: Andreas BEXELL
Supervisors: Patrik BANNURA,
Dr. Jesper ANDERSSON
Examiner: Dr. Johan HAGELBÄCK
Semester: VT 2017
Subject: Computer Science

Abstract

Java has recently become a multi paradigm language, with the functional paradigm now made available alongside the traditional, imperative, one. Programming in the functional paradigm may be considered complicated or hard to read, and there may be concerns that the code it produces may become hard to maintain because of complexity or readability issues. On the other hand, proponents of the functional paradigm promises smaller amounts of less complex code, as the framework takes on a larger responsibility. This Bachelor's thesis closely examines the differences between effectively equal code written in functional and imperative Java, respectively, from the aspects of readability, complexity and verbosity, and shows that while code written in the functional paradigm is smaller and less complex, it is not harder to read.

software architecture, java, functional java, complexity, readability, verbosity, functional programming

Contents

1 Introduction

Java has recently made the transition from being an imperative language and become a multi-paradigm language. This is a transition that has become quite common, with JavaScript and Python as notable examples of modern multi-paradigm languages.

Adopting functional programming into the Java ecosystem consequently enables a new style of programming applications and interact with frameworks and API:s, potentially creating a development process that is easier and faster, since functional programming frameworks aims to lift the programmers work to a higher level of abstraction, making the framework and the execution environment take more responsibility, and even potentially programs that execute faster and safer, as the style of programming becomes simpler.

This thesis for Bachelor of Science in computer science for 15 HEC spring 2017 tries to measure the cost of maintaining functional Java code, as compared to maintaining imperative Java code, by trying to measure relative readability, complexity and verbosity.

1.1 Background

This chapter provides context for the thesis, including a brief introduction of the concepts used.

1.1.1 Maintainability, readability, complexity and metrics

After initial development and delivery, code often go into a long maintenance period, during which bugs are fixed, new features are added, and the code is adapted as a consequence of changes in its environment and what other systems it interacts with. This phase can be very long, and often a considerable part of the time and effort spent on a program is in this phase. This is sometimes overlooked or forgotten in research and cost predictions of software systems [1].

To be able to effectively maintain software it is important that it is well understood by the maintainers. Effective code maintenance requires not only programming skills, but also a thorough understanding of the code and what it does. The maintainer can be the same as the initial programmer, but more often it is not, and even so, it can be surprisingly hard to understand one's own code after a few weeks or years. To create code that it possible to maintain efficiently, it is important that the code is readable.

An important part of readability is layering abstraction. This is one of the reasons *functions* (in the sense of *procedures*) where invented, apart from reuse. To be able to name a number of instructions, larger programs could become more readable, as the details of the implementations where hidden behind a descriptive name. Another layer of abstraction is *Object Orientation*, where functions are grouped together in units according to a model that is supposed to be easy to understand. This way, we can focus our attention to the parts that are immediately interesting, and leave the other parts alone.

When it comes to functions, it was quickly realized that a function that does too much is hard to read and understand. This was formally thought of as *complexity* and there where early notable work to try to formalise and measure complexity, notably McCabe [2] and Halstead [3]. Complexity measures often counts the number of operations and operands that are in play in the same context, or the number of possible execution paths. Today, it is not uncommon that companies have thresholds for how much complexity,

according to some measure, that is allowed in a function or a class, and complexity measurement is often automatic.[1]

Complexity affects readability as well as testability and therefore has an impact on maintainability, but readability is not only determined by complexity. Readability is also affected of the formatting of the code, and the way abstraction is handled - well defined, designed and named functions tend to be easier to read. Therefore, when trying to measure readability, the complexity measure is not enough.

1.1.2 Imperative programming

Imperative programming is a style of programming that is probably the most commonly known. It focuses on writing statements that are executed in order by that machine, where each statement can change the state of the program. The statements jointly makes up a recipe for *how* a computation should be carried out, and fixes the problem solution to the implementation. This gives ultimate control and responsibility of the problem solving to the programmer.

Consider the following code:

Listing 1.1: Imperative implementation of sum array

```
public int i(int[] ints) {
  int s = 0;
  for (int i : ints) {
    s+=i;
  }
  return s;
}
```

The program is meant to produce the sum of an array of integers. It describes the algorithm to do so: take the integers, one by one, starting with the first, moving on to the second and so on, and add them to a local variable containing to sum. When all of the integers in the array have been visited, the sum is returned.

Traditional examples of imperative languages include C, PHP and Java.

1.1.3 Functional programming

Functional programming is meant to allow the programmer to work on a higher level of abstraction than in imperative programming. In functional programming, functions are thought of in the mathematical sense, rather than subroutines, and they cannot have side effects - that is, alter the state of the program. The functional programming paradigm is heavily influenced by lambda calculus.

Functional programming allows the programmer to a higher degree to express *what* should be done, and leave *how* to the framework.

Consider the following code:

Listing 1.2: Functional implementation of sum array

```
public int j(int[] ints) {
  return Arrays.stream(ints).sum();
}
```

[1] This has lead to some complexity measures, while good, have been more or less abandoned, simply because they are hard to automate.

While it may seem like the function is just delegating the computation to another function (sum), that is not all of what happens. In reality, the int array is converted to a *Stream*, which is one of the new constructs in Java 1.8 that has been added to support functional programming. A *Stream* is similar to an Iterator, in that it goes though the items it contains one by one, but differs from an Iterator in that it is open-ended and may contain an infinite number of elements. The other difference is more subtle: it supports a number of transformation operations, like *filter*, that lets through only elements fulfilling a certain criteria, and *map*, that transforms the elements in the stream. Each of these operations formally return a new stream, but in reality they are lazy and stored in the framework until a result is needed. They can then be logically reordered to create an equal but more efficient expression, that is performed once elements from the resulting stream are consumed. This can be done in sequence or in parallel on different cores. All of this is handled by the framework, and hidden from the programmer - who have described *what* needs to be done, rather than *how*.

In the, perhaps trivial, case of summation, we don't know if the stream is processed forwards or backwards, or in one or many cores. All of this is left to the framework to decide.

Traditional examples of functional languages are Haskell, Erlang and Lisp.

1.1.4 Multi-paradigm languages

Traditionally, the functional paradigm and the imperative and object oriented paradigms have operated in different languages. Recently it has become common for traditionally imperative languages to incorporate elements of functional programming, becoming multi-paradigm programming languages, either through new frameworks, or through features being added to the languages themselves. This, ideally, creates programs that are simpler, yet more efficient, while letting the programmer choose abstraction level fitting a certain task dynamically.

Recent examples of multi-paradigm languages include Python, Ruby and Java.

1.1.5 Jumping paradigms

To a programmer trained in the imperative paradigm, though, the functional paradigm may present a steep learning curve. Learning functional programming is not about learning a new framework or API, but rather about leaning a new way to think about problems and solutions. To the imperative programmer, learning functional programming may initially be confusing and its evangelists may seem zealous.

To overcome this initial hurdle, education, training and exercise is needed. This is expected, but how can this be managed on a large scale? For an organisation to be efficient, a broad common understanding among its agents is imperative. Is this, from an organisation's point of view, effective? To answer that question, an analysis of the benefits of switching paradigm is needed.

A large part of the cost of software development is in its maintenance. If the maintenance can be made more efficient by using functional methods, the effort might be worthwhile. The effort of maintaining software depends on how much there is, how complicated it is and how readable it is.

1.2 Related work

There are many studies of complexity of software, how to measure it and what implications it has. Most famous, perhaps, is the Cyclomatic Complexity measure, as proposed by McCabe [2], who counted the number of different paths though a part of code, in an effort to measure how many test cases was needed to cover the code, but there where other early work, such as Halstead [3], who proposed a measure of different aspects of the code and combined them into measures related to complexity, readability, effort and buggyness. Nejmeh [4] proposed a broader complexity measure called *NPATHS*, counting not only branches, but also invocations, statements and expressions towards complexity. This relates to the Volume of the Halstead measure.

Van den Berg [5] did work on readability and complexity of functional programs. He proposes a method of measuring cyclomatic complexity based on a count of reused pattern identifiers and non-identifiers, added to the count of guards, operators, filters and the complexity of list comprehensions. The measure is designed to be comparable to the McCabe cyclomatic complexity metric for imperative languages, but does not make reliable cross-comparisons reliable.

Coplien [6] wrote about multi-paradigm design, and found that being able to use more than one paradigm allows for a better fit between program and domain, as well as a better fit between program and human psyche.

Ryder [7] explores the usability of software metrics in functional code. He points out that there is a wide body of research and tools available on metrics on imperative and object oriented development, but that development in functional languages currently lack a widely adopted software engineering process, and that there has been little research on the topic [7, p 254].

Buse and Weimer [8] built a model for machine learning to measure software readability, but fail to test their model subjects - meaning that the model accounts only perceived readability, but not to what extent the subjects actually understand what they have read.

Håkansson and Badran [9] validate the Van den Berg-method, originally developed for Miranda, but adapted to Haskell by Ryder [7], on functional JavaScript. This makes the measure easily adaptable to Java. They also point out that a study of the relative complexity of imperative versus functional code using the Van den Berg method would be useful.

Glass [1] lists fundamental facts from vast experience and research about software development, where the most relevant ones for this paper are "Maintenance typically consumes about 40 to 80 percent (60 percent average) of software costs. Therefore, it is probably the most important life cycle phase." and " Learning a new tool or technique actually lowers programmer productivity and product quality initially. You achieve the eventual benefit only after overcoming this learning curve."

Regarding verbosity, Lee [10] has made a tool to measure the size of source code.

1.3 Problem formulation

The introduction of functional programming into Java potentially creates a multi paradigm environment for programs. This potentially creates a greater and more flexible toolbox to Java programmers. These new tools are powerful, but may seem awkward and unfamiliar, and there are doubts about what cost they will introduce in the maintanence phase. Given the choice between functional programming style and imperative, which should one prefer? It is the role of software architects to guide developers in this decision, but to properly do so, knowledge is needed. Is one style better from a given point of view than

another? Can we expect increasing, or decreasing, cost of a software system, counted over the course of its lifetime?

1.4 Motivation

Functional programming is still largely unknown amongst Java programmers, used to the strictly imperative object oriented environment that was the norm of Java until version 1.8. Is it worthwhile for a programmer to learn a new paradigm? [1, T2]

It is the role of a software architect to issue and enforce processes and guidelines in a software development organisation. The software architect deals in what is often called non-functional requirements. These are sometimes reffered to as "-ilities", due to the nature of how to mention them: "reusability", "testability", "maintainability", "debugability", "stability" etc.

The introduction of a new paradigm into a given language requires reaction from the software architects in charge of systems being developed and maintained in that language.

Is it beneficial for an employer to educate programmers in the new paradigm? Should software architects issue guidelines about whether to prefer or avoid functional programming in Java? To be able to answer this kind of questions, it is important to evaluate the value of Java written in the functional paradigm.

1.5 Objectives

Q Is functional Java better?

To be able to answer this, there is a need to understand and try to foresee the cost of production and the future maintenance cost of a program or system, and how this is affected by the paradigm used. Maintenance typically accounts for 40-80%, averaging 60%, of software development cost [1]. Focusing on maintenance allows us another question:

Q Is functional Java more maintainable?

Measuring absolute maintainability of a program is the holy graal of software architecture. Maintainability is the common goal of many of the other "-ilities" that the software architect works with. This bachelor's thesis will not try to conjure up an absolute measure of software system maintainability, but rather try to measure relatively between small snippets of imperative and functional Java from different aspects that has a profound impact on maintainability.

To enumerate and be able to examine these aspects, the following questions demand answers:

RQ1 Is functional Java easier to understand than imperative Java?

RQ2 Is functional Java less complex than imperative Java?

RQ3 Is functional Java less verbose than imperative Java?

It is expected that the study will show that functional Java is relatively easier to understand to people familiar with functional programming, and that programmers who are not familiar with functional programming will struggle to understand the functional Java, but not significantly so. The functional programming style generate shorter idioms, allowing for faster recognition by subjects trained in these idioms. Therefore, functional Java

should be faster to read, compared to imperative Java, to programmers experienced in the functional paradigm.

Functional Java is expected to be less or comparably complex compared to imperative Java. The functional programming tools are designed to allow the programmer to operate on a higher abstraction level, thus encompassing more of the complexity involved in common tasks to a greater degree than imperative APIs can do. If this is successful, complexity should move out of the program implementations and into the functional framework, and a consequence would be that the implementation of functional programs contain less complexity than the equivalent imperative implmentation.

Functional Java should be less verbose, than imperative Java. As complexity and implementation details move out of the program, and the program itself becomes more abstract and less complex, the amount of code to complete a task should also become lower. This will allow for creating of code that is easier to maintain, simply because there is less of it.

1.6 Scope/Limitation

Measuring relative complexity between different paradigms is hard, and there is currently no widely accepted metric to do so. This paper will display the results from different metrics, but developing a metric by which programs of different paradigms can be safely compared to each other is beyond the scope of this paper.

Development of programs accounts for approximately 40% of the cost of software development (the 60% typically spent in maintenance) [1, M2]. Measuring the comparative speed of development is not within the scope of this paper. The speed of development is affected by a vortex of parameters, including the quality of the requirements, the processes being used, the tools, the expected quality of the resulting code, to mention a few. To isolate paradigm as a factor is impossible.

An important aspect of maintainability, beside complexity and readability, is debugability, but that aspect is left out of this study. Debugability is dependent on tools and thus might change rapidly with the introduction of a new tool chain. It is also unfair to measure relative debugability without also measuring relative buggyness - if one paradigm is less prone to error, then the debugability might be less important. All of this is left outside the scope of this study.

1.7 Target group

In order to properly guide and lead a development organisation and to make the right decisions, software architects and software managers need information and knowledge about the options and their consequences. This information and knowledge gives confidence in decision making and serves as a tool for gentle persuasion if needed.

This investigation aims to provide a basis for decisions about whether to adopt or reject the functional paradigm in Java, and determine if an education effort is sustainable. It creates and catalogues new knowledge about the nature of functional Java and how it compares to traditional imperative Java.

As such it is of interest to individuals and organisations working with Java development and maintenance.

1.8 Outline

The *Method* chapter contains detailed information about how this study is desinged and carried out. The study is broken down into three sub parts, covering readability, complexity and verbosity. The chapter will elaborate on how to properly evaluate readability, discuss different methods of complexity analysis and how to apply them, and cover how to measure verbosity properly in a multi paradigm environment. It will also account for the examples of idioms used throughout the paper, that come in a imperative and a functional version. It also contains discussions about the reliability and validity of the study and about ethical considerations that have been taken into account when conducting the interviews for the readability survey.

The *Results* chapter enumerate the results from the different measurements carried out according to the methods chosen and covered in the methods chapter. It is a rather brief chapter, presenting the the results while not going in to analysis, which is the topic of the next chapter.

The *Analysis* chapter contains analysis of the results for the different aspects measured in the study. It thorougly covers the results, interprets them, and elaborates on different ways to evaluate them. It also contains an analysis of the validity of the results and their significance.

The *Discussion* chapter revisits the three aspects and provide the answers to the research questions "Is functional Java easier to understand than imperative Java?", "Is functional Java less complex than imperative Java?" and "Is functional Java less verbose than imperative Java?". It also combines the results and analysis of the aspects of readability, complexity and verbosity into a compound analysis and a provides a discussion about the results and how to interpret the value of the different aspects combined.

The *Conclusion* chapter summarizes the conclusions that should be drawn from this study, and gives the target group, software architects and managers, the basis to make informed decisions about how to relate to the introduction of the functional programming paradigm into Java. It also comments on possible future work in the field.

Following the main body of the text is a list of *References* and a few appendices: *Appendix A* is the interview guide for the readability study; *Appendix B* is a complete listing of the idiomatic snippets used; and *Appendix C* is the implementation of the script making a branch-and-invocation count on disassembled Java byte code.

2 Method

To be able to do a controlled comparative study of different paradigms, a number of snippets where chosen and/or produced. These snippets represent idiomatic solutions to identical programs in the two different paradigms, and are compared with regards to readability, complexity and verbosity.

An experiment regarding readability is performed, where subjects are presented snippets of code in a random order. Each snippet presents an idiomatic solution to a problem either in imperative or functional style. The time until the subject can explain the behaviour of the code is measured. This way, actual readability, as opposed to perceived readability, can be measured. Thus, any readability differences between functional and imperative Java can be found. This measurement is used to answer the question "Is functional Java more readable than imperative Java?"

For complexity, three different common complexity measures are applied: McCabe's cyclomatic complexity, Van den Berg's cyclomatic complexity and NPATHS. A complexity measurement tool based on byte code analysis is created and applied as well. This way, any systematic differences in complexity between functional and imperative Java can be found. These measurements can then be used to answer the question "Is functional Java less complex than imperative Java?"

Regarding verbosity, programmatic lines, characters and text lines are counted. Since functional programming typically use very few programmatic lines, it is not an adequate measure. By adding the measurements of characters and text lines, a more complete picture is drawn. These measurements are used to answer the question "Is functional Java less verbose than imperative Java?"

2.1 Measuring readability

Readability is a measure of how fast and well a programmer can understand a piece of source code. As such, a good readability study must measure the time it takes for a programmer to understand a specific piece of code.

The measuring is carried out with guided interviews. After an initial survey of the subjects' experiences, the subjects are presented with the snippets in random order. Time is measured until the subject can correctly explain the functionality of each snippet. The interview guide is available as Appendix A.

The randomization of the order is important to try to counter a "learning effect" where the subject recognize a problem from an earlier example [7, 2.4.2, p. 38].

The time measurements will be correlated against the subjects' experience of programming, and the self-assessment of the programmer's skill in a certain paradigm. This way, the importance of experience in a paradigm can be extrapolated by correlating the timing results against experience of programming and self assessment of proficiency in different paradigms.

Care is taken to select subjects with different experience and skills, from students without professional experience to highly experienced professionals in a large company.

From the interviews, it will be possible to extract information about the relative readability of the different paradigms, as well as to what extent training in each paradigm effects the readability of code in a paradigm. Specifically, how often is a functional paradigm understood faster than its imperative equivalent? This can be counted on the total, but might not be applicable to individuals, where the data might be affected by the learning effect.

By counting the number of times a functional idiom has been understood faster than its imperative equivalent, a measure of the relative readability is created.

2.2 Measuring complexity

The complexity is measured using different metrics, in an effort to counteract any problems a certain measure might have in a certain paradigm.

McCabe's [2] cyclomatic complexity metric is the most widely known and used, and is used in this study. McCabe's metric has been under critique for being too adapted to the imperative style of programming, while giving low scores to code written in the functional paradigm. It is designed to measure the number of paths through a certain piece of code, and thus counts the number of conditional executions.

Functional code is typically much less dependant on branching and explicitly conditional execution. Van den Berg [5] made an effort to use the findings of Halstead [3] to create a cyclomatic complexity metric that would be equally applicable to functional and imperative code. The Van den Berg metric is used in parallel to the McCabe metric of cyclomatic complexity.

NPATHS was designed based on the cyclomatic complexity measure but count more elements towards complexity and is intended to be more exact than McCabe's cyclomatic complexity measure. Since NPATHS count not only branches of the code, but also count for example logical expressions and invocations of functions towards complexity, it is expected to be more applicable to functional code. NPATHS is related to the Halstead Volume, which is why the Halstead Volume is not measured.

There are a plethora of tools claiming and aspiring to measure McCabes Cyclomatic Complexity and/or NPATHS. One of the more popular and widely used in the industry is *checkstyle* [11], which allows measuring of both cyclomatic complexity according to McCabe and NPATHS, but testing of the NPATHS metric running *checkstyle* reveals that *checkstyle* measures the same values for McCabes Cyclomatic Complexity and NPATHS for all of the defined snippets. This is not right, and NPATHS should measure higher in all cases, since more elements are counted towards the complexity. Manual count confirms that *checkstyle* measures McCabes Cyclomatic Complexity correctly.

The complexity study will use *checkstyle* to measure McCabe's cyclomatic complexity, but the other measures will be done by manual count.

Below is a more detailed explanation of each of the metrics and how to count them.

2.2.1 Measuring Cyclomatic Complexity according to McCabe

While McCabes Cyclomatic Complexity measure is based in informatics, measuring the ratio of nodes and edges in a graph representing the program, McCabe also showed that the cyclomatic complexity for a program with only one entry point is equal to the number of decision points, and tools typically count the number of keywords representing a fork in the program flow (in the case of source code analysis) or byte codes representing forks (in the case of byte code analysis).

For an open graph program, the McCabe cyclomatic complexity for a function is the number of decisions points (that is, branches) and operators, plus one.

By convension a number of 1-4 is considered easy to test, 5-7 is OK, 8-10 is considered complex, and over 10 is bad.

Consider the following snippet:

Listing 2.1: Imperative implementation of find an object matching a condition

```
public String a(Collection<String> c, String s) {
  for (String b : c) {
    if (b.startsWith(s))
      return b;
  }
  return s;
}
```

There are two "decision points", namely `for` and `if`, and one is added for the function itself, counting up to a complexity of 3.

2.2.2 Measuring Cyclomatic Complexity according to Van den Berg

Van den Berg [5] created a method of measuring cyclomatic complexity based on a count of reused pattern identifiers and non-identifiers, added to the count of guards, operators, filters and the complexity of list comprehensions. The measure is designed to be an alternative to the McCabe Cyclomatic Complexity metric better adapted to functional languages. Håkansson and Badran adapted this measure from Haskell to JavaScript, in a way that makes it equally applicable to Java [9, Chapter 4.4, p. 30].

Van den Berg makes a distinction between the left hand side, *LHS*, and the right hand side, *RHS* of the expression, where the LHS is the number of pattern identifiers minus the unique pattern identifiers, plus the non identifiers, and the RHS is the guards, the logical oerators, the number of transformations and filters and the pattern complexity of the list comprehensions. The pattern complexity is counted like the LHS. The total complexity is $M = LHS + RHS + 1$.

Consider the following example:

Listing 2.2: Functional implementation of find an object matching a condition

```
public String b(Collection<String> c, String s) {
  return c.stream()
      .filter(b -> b.startsWith(s))
      .findFirst()
      .orElse(s);
}
```

$$LHS = 3 - 3 + 0 = 0 \tag{1}$$

$$RHS = 1 + 1 = 2 \tag{2}$$

$$M = LHS + RHS + 1 = 3 \tag{3}$$

The cyclomatic complexity according to Van den Berg is 3.

2.2.3 Measuring NPATHS

NPATHS [4] counts decision points, gotos, breaks, continue, statements, returns, function calls, statements and operators in expressions. This can lead to a high count, as compared to McCabe's cyclomatic complexity, and a common threshold for a function is 200.

Consider the following example:

Listing 2.3: Functional implementation of Create Histogram

```
public Map<String, Long> h(List<String> c) {
  return c.stream()
       .collect(Collectors.groupingBy(e -> e,
           Collectors.counting()));
}
```

The NPATHS becomes 1 for *return* and one for each function call (4). Total NPATHS become 5.

2.2.4 Measuring branches and invocations on byte code

A conceivable metric concerning complexity, based on byte code analysis would be to count the number of branches, conditionals, and invocations made by a method. This offers an estimate of the complexity of the method as generated by the compiler, and works equally well regardless of paradigm as a consequence of the straight-forward instruction set of the Java byte code, with separate, unambiguous operations for branching and invocations.

A program for such analysis is available in Appendix C. The `.class` file is decompiled using `javap -c`, and the resulting instructions are counted for each function. This makes it trivial to count the number of invocations, the number of conditional jumps (beginning with *if*), and the *return*-statements.

The following Java source code

Listing 2.4: Functional implementation of find an object matching a condition

```
public String b(Collection<String> c, String s) {
  return c.stream()
       .filter(b -> b.startsWith(s))
       .findFirst()
       .orElse(s);
}
```

produces the following disassembled byte code:

Listing 2.5: generated byte code

```
public java.lang.String b(java.util.Collection<java.lang.
    String>, java.lang.String);
  Code:
     0: aload_1
     1: invokeinterface #48, 1  // InterfaceMethod java/util/
        Collection.stream:()Ljava/util/stream/Stream;
     6: aload_2
     7: invokedynamic #55, 0    // InvokeDynamic #0:test:(
        Ljava/lang/String;)Ljava/util/function/Predicate;
    12: invokeinterface #56, 2  // InterfaceMethod java/util/
        stream/Stream.filter:(Ljava/util/function/Predicate;)
        Ljava/util/stream/Stream;
    17: invokeinterface #62, 1  // InterfaceMethod java/util/
        stream/Stream.findFirst:()Ljava/util/Optional;
    22: aload_2
    23: invokevirtual #66       // Method java/util/Optional.
        orElse:(Ljava/lang/Object;)Ljava/lang/Object;
```

```
26: checkcast     #30          // class java/lang/String
29: areturn
```

From this listing, it is trivial to count invocations (5) and returns (1), giving a metric number of 6.

2.3 Measuring verbosity

The most established method of measuring size of source code is lines of code (often measured in the thousands and abbreviated *KLOC*). Lee [10] has created a tool to measure this on Java, with a metric called NCSS.

The problem with only measuring lines of programmatic code is that it is heavily skewed towards functional programming, where functions can be chained to create powerful expressions in a single programmatic line. Such lines can become very long and make a program unreadable if the long programmatic lines is not broken up to several text lines in the source code.

For comparison, the verbosity study also counts the number of characters needed to create the code, as well the number of text lines the idiom will occupy in well formatted source code. What "well formatted code" is, is a matter of opinion and some debate, and will here reflect the author's experience from working with code formatted according to Oracle and Google standards, while not necessarily conforming to either one.

In well formatted code, long lines are broken so that there is one function invocation on each line, and long parameter lists are broken to several lines to increase readability. Broken brackets are treated like in the Oracle standard, with openings on the same line as the declaration and closing on their own line.

2.4 Idioms

To make a comparative study between imperative and functional programming styles, the snippets presented below where created. The snippets come in pairs that are designed to work equivalently and represent examples of solutions of common classes of problems, in a way that feels idiomatically familiar to programmers used to imperative Java and functional programming respectively.

Most examples are taken from refactorisation efforts in a big commercial system, but some are constructed to illustrate certain points. They are all minimized and anonymised.

2.4.1 Find an object matching a condition

This snippet searches through a collection for an entry fulfilling a condition that is a property of the object, rather than identity, and returns that object if found. If not, it returns a default object.

Listing 2.6: Imperative implementation of find an object matching a condition

```
public String a(Collection<String> c, String s) {
  for (String b : c) {
    if (b.startsWith(s))
      return b;
  }
  return s;
}
```

Listing 2.7: Functional implementation of find an object matching a condition

```
public String b(Collection<String> c, String s) {
  return c.stream()
       .filter(b -> b.startsWith(s))
       .findFirst()
       .orElse(s);
}
```

This kind of code is common for selecting the most relevant result from a collection of possible ones, for example finding the path or a language pack of setting package for a specific hardware.

The imperative version, listing 2.6, uses a `for` loop to search through the collection and an `if`-conditional to execute the condition. If a matching object is found, the program makes an early return.

The functional version, listing 2.7, creates a `Stream` of the elements in the collection, and filters that stream to let through only the elements that matches the condition. From this filtered stream, it selects the first element. The `findFirst` method returns an `Optional`, meaning that it returns the value if there is one, or it executes its `orElse`-method, returning the default value.

2.4.2 Create a random String

This snippet creates a String with a given length and random content.

Listing 2.8: Imperative implementation of Create random String

```
public String d(int n) {
  String S = "abcdefghijklmnopqrstuvxyz";
  Random r = new Random();
  String s = "";
  for (int i=0; i<n; i++) {
    s += (char)S.charAt(r.nextInt(S.length()));
  }
  return s;
}
```

Listing 2.9: Functional implementation of Create random String

```
public String c(int n) {
  String S = "abcdefghijklmnopqrstuvxyz";
  return new Random()
       .ints(n, 0, S.length())
       .mapToObj( i -> ""+(char)S.charAt(i))
       .reduce("", (a,b) -> a+b );
}
```

This code is usable to create example strings for testing of sorting algorithms, but also for other types of testing where a large number of strings may be needed.

Both versions rely on a `String` that holds the letters of the alphabet. Letters are randomly selected from the alphabet string.

The imperative version, listing 2.8, creates an object of the class `Random`, and an empty `String`, that is to become the return value. It uses a for loop to execute an instruction the given number of times. Inside the loop, it uses the `Random`-object to

obtain a new random integer and select the letter with the corresponding position from the `String` containing the letters of the alphabet, and concatenates that letter at the end of the `String` that is to be returned. When the `for`-loop exits, the right number of letters have been selected and concatenated and the `String` is returned.

The functional version, listing 2.9, creates a stream of `n` random integers between 0 and `S.lenght()`, to then map them to a stream of `String`s, each string one letter long with the content of the letter corresponding to the integer. This `String`-stream is then reduced with concatenation into a `String` with `n` letters, and returned.

2.4.3 Register callback handler

This snippet registers a callback handler with a conditional execution.

Listing 2.10: Imperative implementation of Register Callback handler

```
public void e(CallbackEvent expected) {
  registerCallback(new CallbackInterface() {
    @Override
    public void callbackMethod(CallbackEvent event) {
      if (expected.equals(event)) {
        doit();
      }
    }
  });
}
```

Listing 2.11: Functional implementation of Register Callback handler

```
public void f(CallbackEvent expected) {
  registerCallback(event -> {
    if (expected.equals(event)) doit();
  });
}
```

This is very common for event listeners like hardware availability callbacks of user interaction callbacks, among many other similar cases.

The imperative version, listing 2.10, creates an anonymous inner class implementing `CallbackInterface`, and explicitly overloads the `callbackMethod`. It uses an `if`-clause to determine if the occurred event matches the expected one, and in such case calls `doit()`. In accordance with traditional Java style guidelines, the `if`-clause has broken brackets ({}), despite being only one line long.

The functional version, listing 2.11, takes advantage of the fact that `CallbackInterface` can be used as a functional interface due to the fast that it has only one abstract method defined. This makes it compatible with Java's lambda expressions, declared with ->. This takes a lot of the boiler plate code of defining an anonymous inner class. Additionally, functional languages are by convention a bit laxer on the requirements of broken brackets and line breaks, allowing us to express the `if`-clause and following `doit()`-call on one line.

2.4.4 Create Histogram

This snippet creates a histogram out of the contents of a List.

Listing 2.12: Imperative implementation of Create Histogram

```
public Map<String, Long> g(List<String> c) {
  Set<String> s = new HashSet<>(c);
  Map<String, Long> m = new HashMap<>();
  for (String e : s) {
    m.put(e, (long) Collections.frequency(c, e));
  }
  return m;
}
```

Listing 2.13: Functional implementation of Create Histogram

```
public Map<String, Long> h(List<String> c) {
  return c.stream()
      .collect(Collectors.groupingBy(e -> e,
          Collectors.counting()));
}
```

The code converts a list containing reoccurring elements into a map where the elements are keys and the values represent the number of occurrences of each element. This is useful when analysing sets of sample data, and is for example an important step when making a block diagram.

The imperative version, listing 2.12, starts with creating a `Set` containing the unique elements of the provided `List`. This `Set` will be equal to the resulting maps key set. This step is strictly not necessary: it is fully possible to iterate over the provided `List`, but the `Set` potentially saves a great deal of runtime. It iterates over each of the elements of the set with a `for`-loop, and saves each element and its frequency in a `Map` that is returned once the loop terminates.

The functional version, listing 2.13, creates a `Stream` of the elements of the `List`, and collects them, grouping by the identity of each of the elements, and uses the counting collector of the `Collectors` class to count the frequency of each item.

2.4.5 Sum an array of ints

This snippet sums all integers in an array.

Listing 2.14: Imperative implementation of sum array

```
public int i(int[] ints) {
  int s = 0;
  for (int i : ints) {
    s+=i;
  }
  return s;
}
```

Listing 2.15: Functional implementation of sum array

```
public int j(int[] ints) {
  return Arrays.stream(ints).sum();
}
```

This simple code returns the sum of the integers in an array. It is for example useful when implementing a command line calculator based on Polish notation.

The imperative version, listing 2.14 creates an integer with the value of 0, iterates through the entries of the array in a `for`-loop, adding each of the numbers to the local integer. Then the array is exhausted, the value of the local integer is returned.

The functional version, listing 2.15, creates a stream of the array, and calls the special-case reduction method `sum`, which is an additive reduction, and returns the result.

2.5 Reliability and Validity

There are many methods for measuring complexity of programs, counting different factors towards complexity. To get a reliable comparison of the complexity difference over the paradigm boundary, this paper employs four different methods, namely McCabe's and Van den Berg's cyclomatic complexity measures, NPATHS and byte code branch-and-invocation count.

The most common method of measuring size of source code is by programmatic lines of code. As this is of dubious applicability to functional code, and two additional measures are employed to complete the picture. One measure is counting text lines of "well formatted code", but opinions on what is good formatting, as well as style guides, may vary. Different opinions, or application of other style guides, may affect the result. A count of the number of characters needed are also measured, to get a reading unaffected by taste and style.

The readability results depend heavily on the performance of the subjects being interviewed. Care is taken to find subjects with different skill and experiences, but other studies might come to different conclusions. The number of snippets and the number of subjects are very limited, and a larger study might yield more reliable results.

2.6 Ethical considerations

An interview situation is delicate and the subjects may feel they are under scrutiny rather than the code. It is important to explain that this is not the case. The names of the subjects are not recorded, and thus, all subjects are guaranteed anonymity to the extent that a certain set of answers cannot be traced back to them. The names of the subjects are also not shared.

3 Results

This chapter presents the results of the studies of the readability, complexity and verbosity of the snippets chosen for the study.

The first part covers the readability study, presenting the experience, self assessments and the results of the timed readability assessment of each of the snippets.

The second part presents the results of the complexity studies, with the complexity metric numbers for each of the chosen complexity measuring methods.

The third part presents the results of the verbosity measurements, with programmatic lines, number of characters and number of text lines in source code.

3.1 Readability

The readability study yielded the results presented in table 3.1.

	S 1	S 2	S 3	S 4	S 5	S 6	S 7	S 8	S 9	S10
Programming experience	33	22	22	26	30	20	5	8	14	22
Professional experience	27	17	17	20	15	13	0	0	5	12
Self assessment										
Imperative programming	5	5	5	4	3	5	2	3	5	4
OO programming	5	5	5	4	3	3	3	4	5	4
Functional programming	2	2	3	2	3	2	1	3	3	2
Java	4	4	5	4	3	3	3	4	5	3
Reading times										
a: Object, imperative	60	55	33	22	15	41	41	25	24	28
b: Object, functional	64	75	23	32	45	19	35	30	16	47
d: Random, imperative	94	97	50	42	34	22	41	17	13	49
c: Random, functional	80	147	151	33	23	11	60	303	85	61
e: Callback, imperative	88	114	20	48	3	12	56	47	41	53
f: Callback, functional	75	35	42	14	26	11	22	12	10	15
g: Histogram, imperative	185	172	65	68	23	8	60	55	33	97
h: Histogram, functional	52	240	62	94	8	59	44	40	24	40
i: Sum, imperative	18	26	6	12	4	18	23	13	13	14
j: Sum, functional	14	33	6	24	8	7	9	5	9	17
$P(t(f) \leq (t(i))$	0.8	0.2	0.4	0.4	0.4	0.8	0.8	0.6	0.8	0.4

Table 3.1: *Readability study results. Programming experience is the number of years since the subject first learned to program. Professional experience is the number of years the subject has been programming professionally. Self-assessments are on a scale of 1-5, inclusive. Reading times are measured in seconds. Added in the bottom is the probability that the subject has read a functional idiom faster or in equal time to an imperative idiom. The full individual results, with the presentation order of each idiom, are recorded in Appendix A.*

Each of the subjects where asked a number of initial questions about their previous experience of programming and their own assessment of their skills in different areas. The full interview guide is available in Appendix A. The subjects were then presented with the snippets, one at a time, in random order, and asked to explain the functionality of each snippet. Time was measured until the subject could satisfactorily explain the code in each

snippet to the interviewer, one at a time. The same interviewer was used for all interviews, in order to secure that the level of the "satisfactory explanation" was consistent.

Apart from the answers to the questions about experience and self assessment, and the number of seconds each subject took to give an explanation of each snippet, the number of times a subject read a functional idiom in equal or lower time than the same subject read the equivalent imperative idiom is recorded, and presented divided by 5 to give a probability. While this number might not be a great indicative neither of the readability of the idioms, nor of the performance if the individual, due to the learning effect discussed in the *Methods* chapter, it clearly shows that there is a wide spread between the subjects. This is further discussed in the *Analysis* chapter.

3.2 Complexity

The results of the complexity measurements are presented in table 3.2. The McCabe Cyclomatic Complexity is measured by *checkstyle* [11], while the Van den Berg is manually counted using the method described by Håkansson & Badran [9, 4.4, p. 39]. NPATHS is also counted manually, and the branch-and-invocation count is generated with the script given in Appendix C.

	McCabe CC		VdB CC		NPATHS		BnI	
Idiom	Imp.	Func.	Imp.	Func.	Imp.	Func.	Imp.	Func.
Object	3	1	5	3	5	5	9	6
Random	2	1	6	4	12	9	11	8
Callback	2	2	3	3	5	4	3	3
Histogram	2	1	3	2	5	5	11	6
Sum	2	1	2	1	4	3	3	3

Table 3.2: *Complexity according to McCabe, Van den Berg (VdB), NPATHS and branch-and-invocation (BnI) count, for imperative (Imp) and functional (Func) idioms respectively.*

The result clearly shows that the functional idioms give lower complexity counts regardless of the metric being deployed. An interesting side note is that while the scale of the metrics differ, the relative difference between the imperative and the functional code is very similar. This suggests that they in fact measure similar things and that they might provide equally good measures of complexity. [2]

3.3 Verbosity

The results of the verbosity study is presented in table 3.3. NCSS[10] is counted using *checkstyle* [11], while characters and text lines are counted with the UNIX command line tool `wc` for each of the snippets.

It is readily apparent that all of the snippets produce fewer lines, both programmatic lines and well formatted lines in the source code, in the functional version. Nearly all of the functional versions also need fewer characters - the exception being the idiom to select

[2] This, in turn, suggests that further validation of complexity metrics might be needed, but this is beyond the scope of this bachelor's thesis.

	NCSS		Characters		Text lines	
Idiom	Imperative	Functional	Imperative	Functional	Imperative	Functional
Object	5	2	131	139	7	6
Random	7	3	206	195	9	7
Callback	8	4	219	119	10	5
Histogram	6	2	214	145	8	5
Sum	5	2	92	67	7	3

Table 3.3: *Verbosity measured by programmatic lines (NCSS), Characters and Text lines in source code for imperative and functional idioms, respectively.*

an object with a certain property from a list, in which case the functional version need slightly more characters, mainly because of the names of the methods being employed.

The statistical significance of the difference of the measures will be discussed in the *Analysis* chapter.

4 Analysis

In this chapter, the results of the studies regarding readability, complexity and verbosity are analysed and their significance is established.

4.1 Readability

The subjects are generally quite confident in their abilities, with median self assessment in imperative programming, object oriented programming and Java at 4.5, 4 and 4, on a scale of 1-5, respectively. They are, according to the self assessment, not very familiar with functional programming, landing on a median of 2, with no one assessing themselves over 3.

This is not reflective of their results, however. The general probability that a functional idiom is recognised and understood in equal or faster time for all of the samples and subjects is

$$P(t(f) \leq t(i)) = \frac{28}{50} = 0.56 \tag{4}$$

suggesting that there is a small probability of faster reading of the functional idioms, generally. The probability is so close to 0.5, though, that it cannot be considered conclusive.

The notion of inconclusiveness is reinforced by performing an unequal variances two-tailed t-test on the data. For the idioms for finding an object (listings 2.6 and 2.7) the t-test gives a $p \approx 0.59$. For the creation of a random string (listings 2.9 and 2.8), $p \approx 0.12$, the callback handler idioms (listings 2.10 and 2.11) has a difference significant at $p \approx 0.10$, the histogram (listings 2.12 and 2.13) at $p \approx 0.72$ and the summation of an array (listings 2.14 and 2.15) at $p \approx 0.68$. This means that none of the differences are statistically significant at the $p \leq 0.05$ level.

Interestingly, self assessment in functional programming, professional experience or total programming experience over the median all seem to *lower* the probability of decoding functional idioms faster.

A lower than median self assessment of proficiency in functional programming gives a relatively high probability that the functional idioms will be read faster:

$$P(t(f) \leq t(i))_{sa(f) \leq \tilde{sa(f)}} = 0.8 \tag{5}$$

The subjects with lower than median professional experience also read functional idioms faster:

$$P(t(f) \leq t(i))_{pex \leq \tilde{pex}} = 0.68 \tag{6}$$

Even when splitting the subjects on total programming experience, the subjects with lower than median experience seem to be slightly faster in reading functional idioms:

$$P(t(f) \leq t(i))_{tex \leq \tilde{tex}} = 0.57 \tag{7}$$

Two students in the study, with small experience and modest self assessment in functional programming, where to a higher degree relatively faster at reading functional code, as compared to imperative code, than the average in the full sample population. Even disregarding these samples, though, the effect still seem to be that less professional experience correlates to a higher probability of understanding functional code faster.

This might be the result of functional code being more intuitive to programmers with less experience, or might be an effect of curriculum changes to include more functional

programming in educations. It may even be a chimera caused by the limited sample set. Looking at the scatter plot (figure 4.1), this seems more likely. Seldom has a scatter plot been more scattered. There is, however, no data in this study to indicate age or time span in education of the subjects, and more studies are needed to investigate this.

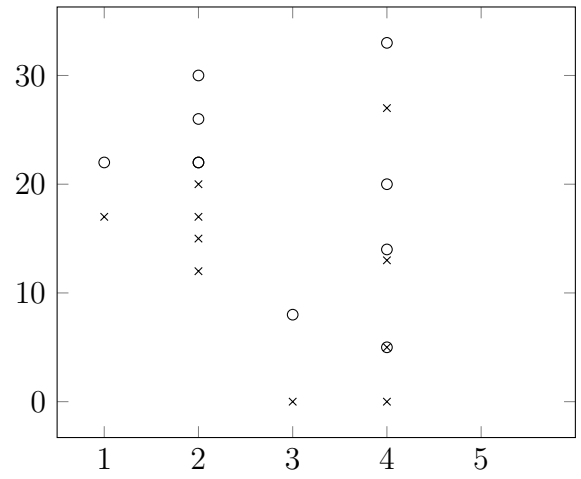

Figure 4.1: *Scatter of professional experience (x) and total experience (o) against number of functional idioms read faster*

Regarding the learning effect, there is some evidence in the results that it has occurred, but also that the randomization of the order in which the idioms have been read affects the paradigms equally, and thus has negligible effect on the total outcome of the study. It does have an effect on individual surveys, and that is one of the reasons it is hard to break down the results into smaller groups.

Generally, there is a small indication that functional code might be easier to read, but the difference is so small that it might be within the margin of error. There is, however, nothing to suggest that reading functional Java is significantly slower than reading imperative Java.

4.2 Complexity

Regardless of the metric chosen - McCabe's cyclomatic complexity, Van den Berg's complexity, NPATHS or branch-and-invocation count - the functional implementation is equally or less complex than the imperative one. The difference is not great in each of the examples, but consider an example program with all of the example methods: the total McCabe cyclomatic complexity metric drops 45%, the Van den Berg metric with 30%, total NPATHS 16% and branch-and-invocation almost 30%.

The change is statistically significant, according to Student's t-test, for the McCabe metric, at $p \approx 0.03$, the Van den Berg metric, at $p \approx 0.02$ and branch-and-invocation count, at $p \approx 0.04$, but not for the NPATHS metrix, at $p \approx 0.07$.

Functional programming aims to take more complexity out of the hands of the programmer and into the framework, and therefore this outcome is expected. It is also noteworthy that the Van den Berg metric, as well as the branch-and-invocation count, both designed with functional programming clearly in mind, drops considerably when implementing the functionality in a functional way.

4.3 Verbosity

In all cases, save one, is the functional implementation less verbose, than the imperative implementation. Counted by programmatic lines the functional implementation is denser in all cases. Totally, the programmatic lines drop with 59%, the number of characters drop with 23% and the number of lines in the source code drop 36%.

The change is statistically significant, according to Student's t-test, for the count of programmatic lines, at $p \approx 1,2 * 10^{-4}$ and well formatted lines, at $p \approx 0.01$, while the count of characters is not, at a $p \approx 0.26$.

That the programmatic lines drop significantly is expected, as functional statements are often "chained" and multiple invocations can be done in a single line, but the number of characters as well as the number of text lines in the code, that are not so heavily rigged against imperative programming, both drop radically as well, clearly showing that functional code is significantly less verbose than the imperative equivalent.

5 Discussion

In this chapter, the results and the analysis of the results are summarized and discussed. First, the three aspects of this study are discussed separately, with answers provided to each of the research questions. After that, there is a discussion about what impact the results, taken together, might have on general maintainability of source code in an environment supporting both the imperative and the functional paradigm.

5.1 Readability

The readability study suggests that functional code may be easier to read, but the difference is so small that it is not possible to make such an assertion. The sample set is small enough that when breaking it down in groups, the skills of the individual subjects have too large impact on the results.

There are artefacts in the readability study, and some of the results seem dubious. It seems that reading functional programming is generally faster, but to what extent is not extractable from the data. The expected effect, that programmers with experience only from imperative programming would struggle with the functional idioms, while programmers versed in functional programming would excel is not detectable in the data.

From the data it is not possible to give a certain answer to *RQ1* "Is functional Java easier to understand than imperative Java?", but there is also nothing to suggest that the functional style of programming is harder or slower to read than the imperative style.

5.2 Complexity

There are many more or less established ways to measure complexity, and this paper employs a few different ones. Some, like the McCabe cyclomatic complexity metric, has been criticised for not being readily applicable to functional code, while others, like the Van den Berg cyclomatic complexity metric, has been designed with functional code in mind. The result of the study is clear: functional code is equally or less complex than imperative code when implementing the same functionality, regardless of what metric is used to measure complexity.

This is to be expected: the goal of the functional paradigm is to raise the level of abstraction, giving the programmers more advanced building blocks and more flexible ways of using them. This leads to more complexity being implemented in the framework, with less complexity left to implement in the program.

RQ2 can thus be answered: Yes. Functional Java is less complex than imperative Java.

5.3 Verbosity

The result of the verbosity study is clear as well: with the functional paradigm providing more flexible and advanced tools, and the programmer working on a more abstract level, less code needs to be written to implement a certain functionality.

Measured in programmatic lines, the difference is titanic: the functional code has less than half the lines of the equivalent imperative code; but programmatic lines is not a good measure, since functional code naturally chains statements on the same programmatic line, and thus creates code that is wide rather than long.

Measured in bytes, the difference is smaller, and in one case the functional code is actually slightly larger than the imperative code. The general difference is about 23%, but the measure fail Student's t-test and does not have statistical significance.

The measure that makes most intuitive sense, although it is dependent on subjective opinion, is how many text lines well formatted code occupies. This is a good measure on "how much there is to read". In this way of measuring, the functional code occupies about two thirds of the lines of the imperative equivalent.

RQ3 can thus be answered: Yes. Functional Java is generally less verbose than imperative Java.

5.4 Maintainability

Combining the results of the different aspects of this study; readability, complexity and verbosity; clearly shows that when comparing functional to imperative Java, the functional Java can produce smaller amounts of less complex code while implementing functionality equal to the imperative code. This is promising, but to any seasoned software architect it should invoke memories of early encounters with Pearl [12], IOCCC [13] or even BF [14], all examples of dense code that sacrifices readability. Promises of less verbose more powerful code make a developer or an architect suspicious.

For exactly that reason, it is important to measure the readability. This study shows that while the functional paradigm in Java do deliver on promises of higher abstraction, less complexity and less verbosity, it also does not become harder to read to an extent that can be measured.

This means that functional style Java can be used to make programs that are likely to be more efficient to maintain.

6 Conclusion

The readability study shows no significant difference in the readability of functional and imperative Java. Interestingly, this is regardless of the self assessment of former experience of functional programming. This suggests that functional-style Java code is about equally readable as compared to imperative Java, to beginners and experienced developers alike.

The complexity analysis shows that regardless of the metrics being applied, functional Java produces less complex code than the equivalent imperative code. This is expected, as one of the goals of functional programming is to increase abstractions and reduce complexity.

The analysis of verbosity shows that functional code is generally less verbose than the imperative equivalent, especially with regards to programmatic lines, but also in terms of text lines in well formatted code. The number of characters needed to express the functionality in the functional paradigm also seem to drop generally, but the difference cannot be statistically established.

Taken together, this shows that functional Java requires smaller amounts of less complex code to implement common functionalities, as compared to imperative Java. This is in line with the goals of functional programming and was expected. Small amounts of simple code is likely to be easier to maintain than large amounts of complex code, provided the developers responsible for the maintenance can properly read and understand it.

This suggests that recommending and training developers to use functional programming can lead to a programming style that produces smaller amounts of less complex code at no penalty to readability. This is likely to mean that maintenance is made easier.

6.1 Future work

Further readability studies are needed to validate the findings and find new data to correlate the readability to. The idiom base and the population should both be larger. The interviews could to some extent be automated and thus be performed on a greater scale. A web based survey that measure the time for reading idioms with the answers being multiple choice could be automated and performed on a scale several orders of magnitude larger than the study presented here, and would be helpful to gauge the readability of different styles of code.

The different metrics to indicate complexity employed in this study showed remarkably similar relative results. Generally, further work on good complexity measures, how to automate their measurements, and how they relate to readability and maintainability may still be needed, even after 50 years.

A larger study of the relative verbosity of the paradigms might also be useful. This study finds significant differences with regards to lines of code, but not in the regard of characters needed to write the code. Further verbosity studies might be able to establish such differences.

References

[1] R. L. Glass, "Frequently forgotten facts about software engineering," *IEEE Software*, vol. 18, pp. 110–112, 2001.

[2] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, pp. 308–320, December 1976.

[3] M. H. Halstead, *Elements of Software Science*. Elsevier North-Holland, Amsterdam, 1977.

[4] B. A. Nejmeh, "NPATH: a measure of execution path complexity and its applications," *Communcations of the ACM*, vol. 31, pp. 188–200, February 1988.

[5] K. Van den Berg, "Software measurement and functional programming," Ph.D. dissertation, University of Twente, 1995.

[6] J. O. Coplien, "Multi-paradigm design," Ph.D. dissertation, Vrije Universiteit Brussel, 2000.

[7] C. Ryder, "Software measurement for functional programming," Ph.D. dissertation, University of Kent, 2004.

[8] W. R. Weimer and R. P. Buse, "Learning a metric for code readability," *IEEE Transactions on Software Engineering*, vol. 36, 2010.

[9] J. Håkansson and S. Badran, "Evaluating cyclomatic complexity on functional javascript," Bachelor Thesis, Linnaeus University, 2016.

[10] C. C. Lee. (2015) Java NCSS. [Online]. Available: http://www.kclee.de/clemens/java/javancss/

[11] O. Burn. (2017) checkstyle. [Online]. Available: http://checkstyle.sourceforge.net

[12] L. Wall. (1987) Pearl. [Online]. Available: https://www.pearl.org

[13] International Obfuscated C Code Contest. [Online]. Available: http://ioccc.org

[14] U. Müller. (1993) Brainfuck. [Online]. Available: http://www.muppetlabs.com/~breadbox/bf/

A Appendix A - Interviews

A.1 Interview guide

- How long since you first learned to program?
- How long have you been programming professionally?
- What is your "main" programming language?
- On a scale from 1 to 5, inclusive, how familiar are you with
  - imperative programming
  - object oriented programming
  - functional programming
  - Java

A.2 Answers

A.2.1 Subject 1

Programming experience	22 years
Professional experience	17 years
Main language	C++
Imperative programming	1 2 3 4 **5**
Object Oriented programming	1 2 3 4 **5**
Functional programming	1 **2** 3 4 5
Java	1 2 3 **4** 5
I Sum (imperative)	18 sec
F Callback (functional)	75 sec
G Histogram (imperative)	185 sec
A Object (imperative)	60 sec
D Random (imperative)	94 sec
B Object (functional)	64 sec
C Random (functional)	80 sec
H Histogram (functional)	52 sec
J Sum (functional)	14 sec
E Callback (imperative)	88 sec

A.2.2 Subject 2

Programming experience	22 years
Professional experience	17 years
Main language	C++
Imperative programming	1 2 3 4 **5**
Object Oriented programming	1 2 3 4 **5**
Functional programming	1 **2** 3 4 5
Java	1 2 3 **4** 5
E Callback (imperative)	114 sec
F Callback (functional)	35 sec
J Sum (functional)	33 sec
I Sum (imperative)	26 sec
A Object (imperative)	55 sec
B Object (functional)	75 sec
G Histogram (imperative)	172 sec
D Random (imperative)	97 sec
H Histogram (functional)	240 sec
C Random (functional)	147 sec

A.2.3 Subject 3

Programming experience	30 years
Professional experience	17 years
Main language	Java, C
Imperative programming	1 2 3 4 **5**
Object Oriented programming	1 2 3 4 **5**
Functional programming	1 2 **3** 4 5
Java	1 2 3 4 **5**
F Callback (functional)	42 sec
H Histogram (functional)	62 sec
C Random (functional)	151 sec
E Callback (imperative)	20 sec
D Random (imperative)	50 sec
A Object (imperative)	33 sec
G Histogram (imperative)	65 sec
I Sum (imperative)	6 sec
B Object (functional)	23 sec
J Sum (functional)	6 sec

A.2.4 Subject 4

Programming experience	26 years
Professional experience	20 years
Main language	python
Imperative programming	1 2 3 **4** 5
Object Oriented programming	1 2 3 **4** 5
Functional programming	1 **2** 3 4 5
Java	1 2 3 **4** 5
C Random (functional)	33 sec
J Sum (functional)	24 sec
E Callback (imperative)	48 sec
A Object (imperative)	22 sec
I Sum (imperative)	12 sec
H Histogram (functional)	94 sec
D Random (imperative)	42 sec
B Object (functional)	32 sec
G Histogram (imperative)	68 sec
F Callback (functional)	14 sec

A.2.5 Subject 5

Programming experience	30 years
Professional experience	15 years
Main language	Java, go, python
Imperative programming	1 2 **3** 4 5
Object Oriented programming	1 2 **3** 4 5
Functional programming	1 2 **3** 4 5
Java	1 2 **3** 4 5
F Callback (functional)	26 sec
D Random (imperative)	34 sec
C Random (functional)	23 sec
G Histogram (imperative)	35 sec
B Object (functional)	45 sec
A Object (imperative)	15 sec
H Histogram (functional)	8 sec
J Sum (functional)	8 sec
I Sum (imperative)	4 sec
E Callback (imperative)	3 sec

A.2.6 Subject 6

Programming experience	20 years
Professional experience	13 years
Main language	python, C
Imperative programming	1 2 3 4 **5**
Object Oriented programming	1 2 **3** 4 5
Functional programming	1 **2** 3 4 5
Java	1 2 **3** 4 5
E Callback (imperative)	12 sec
J Sum (functional)	7 sec
D Random (imperative)	22 sec
H Histogram (functional)	59 sec
A Object (imperative)	41 sec
I Sum (imperative)	18 sec
B Object (functional)	19 sec
G Histogram (imperative)	8 sec
F Callback (functional)	11 sec
C Random (functional)	11 sec

A.2.7 Subject 7

Programming experience	5 years
Professional experience	- years
Main language	Java
Imperative programming	1 **2** 3 4 5
Object Oriented programming	1 2 **3** 4 5
Functional programming	**1** 2 3 4 5
Java	1 2 **3** 4 5
I Sum (imperative)	23 sec
E Callback (imperative)	56 sec
H Histogram (functional)	44 sec
A Object (imperative)	41 sec
G Histogram (imperative)	60 sec
B Object (functional)	35 sec
F Callback (functional)	22 sec
J Sum (functional)	9 sec
D Random (imperative)	41 sec
C Random (functional)	60 sec

A.2.8 Subject 8

Programming experience	8 years
Professional experience	- years
Main language	Java
Imperative programming	1 2 **3** 4 5
Object Oriented programming	1 2 3 **4** 5
Functional programming	1 2 **3** 4 5
Java	1 2 3 **4** 5
A Object (imperative)	25 sec
C Random (functional)	303 sec
E Callback (imperative)	47 sec
I Sum (imperative)	13 sec
D Random (imperative)	17 sec
B Object (functional)	30 sec
H Histogram (functional)	40 sec
F Callback (functional)	12 sec
G Histogram (imperative)	55 sec
J Sum (functional)	5 sec

A.2.9 Subject 9

Programming experience	14 years
Professional experience	5 years
Main language	Java, python, C
Imperative programming	1 2 3 4 **5**
Object Oriented programming	1 2 3 4 **5**
Functional programming	1 2 **3** 4 5
Java	1 2 3 4 **5**
A Object (imperative)	24 sec
E Callback (imperative)	41 sec
J Sum (functional)	9 sec
H Histogram (functional)	24 sec
B Object (functional)	16 sec
C Random (functional)	85 sec
G Histogram (imperative)	33 sec
D Random (imperative)	13 sec
I Sum (imperative)	13 sec
F Callback (functional)	10 sec

A.2.10 Subject 10

Programming experience	22 years
Professional experience	12 years
Main language	C
Imperative programming	1 2 3 **4** 5
Object Oriented programming	1 2 3 **4** 5
Functional programming	1 **2** 3 4 5
Java	1 2 **3** 4 5
D Random (imperative)	49 sec
A Object (imperative)	28 sec
G Histogram (imperative)	97 sec
I Sum (imperative)	14 sec
E Callback (imperative)	53 sec
J Sum (functional)	17 sec
B Object (functional)	47 sec
F Callback (functional)	15 sec
C Random (functional)	61 sec
H Histogram (functional)	40 sec

B Appendix B - Snippets

```
public String a(Collection<String> c, String s) {
  for (String b : c) {
    if (b.startsWith(s))
      return b;
  }
  return s;
}

public String b(Collection<String> c, String s) {
  return c.stream()
      .filter(b -> b.startsWith(s))
      .findFirst()
      .orElse(s);
}

public String c(int n) {
  String S = "abcdefghijklmnopqrstuvxyz";
  return new Random()
      .ints(n, 0, S.length())
      .mapToObj( i -> ""+(char)S.charAt(i))
      .reduce("", (a,b) -> a+b );
}

public String d(int n) {
  String S = "abcdefghijklmnopqrstuvxyz";
  Random r = new Random();
  String s = "";
  for (int i=0; i<n; i++) {
    s += (char)S.charAt(r.nextInt(S.length()));
  }
  return s;
}

public void e(CallbackEvent expected) {
  registerCallback(new CallbackInterface() {
    @Override
    public void callbackMethod(CallbackEvent event) {
      if (expected.equals(event)) {
        doit();
      }
    }
  });
}
```

```
public void f(CallbackEvent expected) {
  registerCallback(event -> {
    if (expected.equals(event)) doit();
  });
}

public Map<String, Long> g(List<String> c) {
  Set<String> s = new HashSet<>(c);
  Map<String, Long> m = new HashMap<>();
  for (String e : s) {
    m.put(e, (long) Collections.frequency(c, e));
  }
  return m;
}

public Map<String, Long> h(List<String> c) {
  return c.stream()
      .collect(Collectors.groupingBy(e -> e,
          Collectors.counting()));
}

public int i(int[] ints) {
  int s = 0;
  for (int i : ints) {
    s+=i;
  }
  return s;
}

public int j(int[] ints) {
  return Arrays.stream(ints).sum();
}
```

C Appendix C - Counting complexity

To count branches and invocations on a byte code level, the `javap` tool is used to disassemble the `.class` file. The output is then piped through the `comp.awk` script, that outputs the summarized branch and invocation counts per method.

Listing C.1: comp.awk

```
#!/usr/bin/awk -f

/Code:/ {print p}
{p=$0}
/^ +[0-9]+: invoke/ {c++}
/^ +[0-9]+: goto/ {c++}
/^ +[0-9]+: if/ {c++}
/^ +[0-9]+: .?return/ {c++}
/^$/ {print c; c=0}
END {print c; c=0}
```