# Measuring time distribution of engineering test and production code

*Version of May 24, 2013*

Wouter Willems

# Measuring time distribution of engineering test and production code

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Wouter Willems
born in Hellevoetsluis, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

# Measuring time distribution of engineering test and production code

Author: Wouter Willems
Student id: 1274473
Email: wouterwillems10@gmail.com

**Abstract**

Testing software is an important factor in application development. Without proper testing, software can be unreliable and show failures with drastic consequences. There have been attempts to measure how much time is spent on testing but these methods are not accurate as they depend on incomplete or subjective data.

An Eclipse plugin called 'WatchDog' enables us to measure how much time a developer spends on reading and writing test and production code. Three different companies and one open source project took part in an experiment where a total of nine developers were monitored from two to seven months. After the results were collected, these developers were interviewed to explain the measurements.

According to this data set, we noticed that time spent on test code can greatly differ as values from 0% to 67% were measured compared to time spent on production code. Values for reading code vary from 22% to 61% of time spent compared to writing code. This thesis goes into depth on this topic to look for explanations of the values that were measured.

Thesis Committee:

| | |
|---|---|
| Chair: | Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft |
| University supervisor: | Dr. A.E. Zaidman, Faculty EEMCS, TU Delft |
| Committee Member: | Dr. C. Hauff, Faculty EEMCS, TU Delft |

# Preface

This thesis was written for my graduation project for the Master of Science degree at the Technical University in Delft. I am grateful to my supervisor, Andy Zaidman, for supervision, valuable feedback and helping me with writing this thesis. I would also like to thank Alex Nederlof, for being a great support during my master, providing me with great knowledge about programming and having interesting discussions with me about computer science in general. I am grateful to my parents, for giving me the opportunity to enjoy this education and motivating me through the process. Last but not least, I'd like to give thanks to my girlfriend, Maaike van der Veer, for keeping me motivated during my graduation project.

Wouter Willems
Delft, the Netherlands
May 24, 2013

# Contents

# List of Figures

# Chapter 1

# Introduction

Software is everywhere. Software helps us complete our daily tasks, big or small. We rely on apps on our smartphones to show up at important meetings, use the software in our car to arrive at destinations safely and manage our bank accounts via the internet. But what if this software starts to fail? Software failures can have big consequences. For example, in 2012, a software failure in a stock market system ended up in a company losing 440 million dollars [1]. More generally, a 2002 report estimates that the US economy loses up to a $59.5 billion each year due to software failures [2].

Testing helps ensuring that a software system behaves properly by defining the system's expected behavior. Test suites assist developers to maintain or extend software with new features, without the risk of unknowingly damaging features that already existed in previous versions [3]. Today, there is a lot of tooling available to ease the process of creating these tests. This has not always been the case [4]. Even though the use of tools eases the process of maintaining test suites, it is still very time consuming. According to literature, about 50% of a project effort is consumed by all forms of testing [5] [6] [7].

How much time is spent on developer testing is not clear. There have been attempts to extract this information from data like code metrics [7], but these are never accurate as they depend on insufficient or subjective information. To our knowledge, there has not been any research on a developer's time spending that is based on time itself, not implied by other data. The goal of this thesis is to fill this gap with a plugin for the Eclipse IDE [8] called WatchDog. By recording intervals in real-time while a developer is busy programming, we can create an accurate profile that provides insight on how much time is spent on reading and writing test code, compared to how much time is spent on reading and writing production code. We confront the participants in this experiment with these results to explain the measurements and do an exploratory study on these results over time.

## 1.1 Research Questions

This research is based on the following research question:

> **Main research question:** How much time do developers spend on test code relative to production code during application development?

Measuring the time distribution of a developer can produce a reliable profile of a developer's working behavior regarding test and production code. This profile is created using four variables, namely the following:

- Time spent reading production code
- Time spent writing production code
- Time spent reading test code
- Time spent writing test code

Using these variables a developer's time profile is created. To create a developer's time profile, we are not interested in absolute values, but rather in relative values. With this, we wish to answer the following sub questions:

> **Sub question A:** How much time do developers spend reading and writing test code relative to production code?

Besides writing the code that makes up the software, it is a common practice to write unit tests to improve the quality of a software product. We are interested in how much time this consumes compared to maintaining production code. We use interview questions to discuss these results and hope to find relations with the environment a developer has to work in.

> **Sub question B:** How much time do developers spend on reading vs writing code?

Working on code does not always imply the altering of code. For example, when a developer lacks the knowledge of the architecture of a software system, it might be necessary to read up on the code first. We are interested in how much time is spent on reading code, including the explanations of the values we measure.

> **Sub question C:** Does the time distribution view of the developer match with the values we measure?

A developer might have a disformed view of his or her own behavior while programming. We are interested in the reasons of why this would be.

## 1.2 Outline

This paper is structured as follows. The next chapter describes the process of creating a tool to measure a developer's time spending. Chapter three introduces the participating companies and explains how the experiment of monitoring developers is set up. Chapter four discusses the results that were gained from the experiment and introduces an exploratory study to these results over time. Chapter five relates our work to other work in this field. We end this thesis with conclusions and future work in chapter six.

# Chapter 2

# Measuring a developer's time distribution

To answer our research questions concerning the time distribution during application development, we need to gain a view of what certain tasks a developer is undertaking and how much time he or she spends doing that. There are multiple ways of obtaining this information. Runeson et al. [6] gathered similar information by using questionnaires, workshops and interviews. Our plugin, called Watchdog, gathers this information by using the Integrated Development Environment (IDE) itself, collecting data in real time. This data is always objective, in contrast to relying on data that is collected afterwards, which might be incorrect. This section explains what is relevant for WatchDog to record, how it is implemented and why WatchDog is a reliable tool to define how much time was spent on reading and writing test code and production code.

## 2.1 What is a developer time profile?

A developer time profile is created to portrait the developer regarding time spending on reading and writing test code and production code. Imagine yourself to be working on developing a new application. Because you think high quality of software is important, you decide to write unit tests while writing production code in a Test Driven Development [9] [10] manner. Besides writing your own code, a colleague has written some classes as well but he failed to write tests for them. In order to write these tests, you need some time to fully understand his code. Your developer time profile at the end of the day could be something comparable to table 2.1.

| Class name | Reading (min) | Writing (min) |
|---|---|---|
| YourClass.java | 100 | 120 |
| YourClassTest.java | 50 | 60 |
| ColleagueClass.java | 150 | 5 |
| ColleagueClassTest.java | 110 | 90 |

Table 2.1: Developer time profile example

In this example, we spent 250 (100+150) minutes on reading production classes while we spent 125 (120+5) minutes on writing them. For testing, we spent 160 (50+110) minutes on reading and 150 (60+90) on writing. This means that we spent 375 minutes on production code and 310 minutes on test code. We can conclude that this day, we spent 45% of our time on reading and writing test code, relative to the amount of time spent on production code.

**Time Profile Requirements** Obtaining such a view can be done by asking a developer by the end of the day what he has spent his time on. Although this approach is very easy, it is not accurate [5] [6]. One can easily make mistakes when asked for estimations of how a day is spent. A developer can forget events such as getting a cup of coffee, or being disrupted by a colleague asking for help. Besides these obvious errors, estimations are subjective and do not provide exact results. Another reason to not go for such an approach, is the fact that a developer is burdened with the task to fill in some questionnaire about his time distribution every day. To avoid these issues while creating a developer time profile, the following criteria must be met:

**Automated** The profile should be created without the need of help of the developer.

**Accurate** The values in the profile should depict an accurate view of the actual work the developer has done.

**Transparent** The profile should be created without disrupting the developer in any way. Violating this criterion could end up in the developer not willing to participate in the experiment.

**Secure** The creation of this profile may not result in possible information leakage, due to the fact that the experiment runs in companies.

## 2.2 Implementing a developer time profile constructor, WatchDog

To implement an automated Developer Time Profile Constructor, it requires some kind of recording of a developer's work while programming. In order to achieve this, the decision was made to create a plugin for the Integrated Development Environment (IDE) the developer is working in. This way, the plugin can use the IDE events to determine what a developer is doing. Before the implementation of the plugin can commence, we need to know what configurations the three participating companies, TOPdesk [11], the Software Improvement Group [12] and InfoSupport [13], use to develop applications. Without the right support of the used applications, the plugin might be useless and will not be able to record any data at all. The following factors are important for supporting the IDEs:

**IDE of choice** The plugin needs to be written for all IDEs (eg. Eclipse, Visual Studio) of choice, including different versions used.

**Language** The plugin needs to support the programming languages used, to determine if a developer is working on test or production code.

**Testing Framework** To determine the difference between working on test or production code, we have to support the testing framework (eg. jUnit, nUnit) that is used.

All three companies use roughly the same configuration:

- Eclipse 3.6+
- Java 6+
- JUnit 4

The only difference between the configurations is the version of Eclipse [8] used varying from Eclipse 3.6 (Helios) up to Eclipse 4.2 (Juno).

The plugin written for this research is an Eclipse plugin called 'WatchDog', which is open source and made available at GitHub [14]. WatchDog records the developer's time spending of the programming activities without burdening the developer. The data that WatchDog collects is stored locally, so no information is leaving the company until the company decides to hand over the data.

### 2.2.1 Determining activity type, test or production

In order to determine whether a developer is working on test or production code, WatchDog analyzes a file by its content. Files not ending with the '.java' suffix, get recorded with the 'undefined' activity type. Many developers indicate a test class with the suffix 'Test'. For example, MyEmailerTest.java. We feel, however, that this is insufficient. The chance of getting false positives by using this method is high. One can imagine to create a sample production class like 'Test.java' with a simple 'hello world' function to see if the first build

of a project is working. Because the word 'Test' is in this file, this would result in defining this class as a test class, which it is not. Another possibility of a false positive is a test class that misses the 'Test' suffix.

To lower the chances of getting these false positives, we need more information about the file. The most information can be gathered by examining the content of a file itself. Because we are to support JUnit 4, every test method needs to have a `@Test` annotation. If we see at least one of those, we can safely assume this is a test class. However, what if, for some reason, a developer left a `//@Test` comment in his production class? Even though this is unlikely, we do not want WatchDog to define this as a test class, just because that comment includes a test annotation. To fix this, we could look at the Abstract Syntax Tree [15] of the class. However, this could make the plugin too heavyweight resulting in slowing down Eclipse which violates the transparency criterion. Therefore, besides looking at the test annotation, the `org.junit` import is also required. This lowers the probability of getting false positives. The only way a false positive is possible now, is when a developer imports JUnit libraries in a production code file, while making comments with test annotation in them which is unlikely to happen. However, this approach keeps the plugin lightweight.

### 2.2.2 Interval Calculation

Development time is measured and recorded in terms of intervals. A complete interval of a task is the duration of when the tasks starts up until the task ends. Imagine the task of reading a document containing production code for 1 minute, editing it for 2 minutes and finally closing the document. This task can be described using two different intervals. See table 2.2.

| Class name | Document type | Activity Type | Duration |
|---|---|---|---|
| YourClass.java | Production | Reading | 60 |
| YourClass.java | Production | Writing | 120 |

Table 2.2: Example of two intervals in tabular format

The challenge in creating this plugin is determining when such an interval should start and when it should end. Because of the differences in how to determine these values for reading and writing, different approaches are mandatory.

#### Writing

If we want to know when an interval starts for writing a document, we first must define what 'writing' really means. In our perspective, writing in a document means that the goal of the developer is to edit or enhance that document. A developer can only focus on editing one document at a time. Would the focus change to some other document, the first interval should close and a new one should be created.

So how can we determine if a developer starts writing? Because we work with the IDE API, we have multiple interesting events to look for. Each of these were considered. We will briefly explain the useful possibilities listed below.

- Focus/unfocus of workspace windows
- Opening/closing of parts [16]
- Changes in a document

**Focus/unfocus of workspace windows** This event fires when Eclipse is gaining or losing focus. Gaining focus would not mean that a developer is about to write code. However, the unfocus event could be interesting to determine that a developer has stopped writing code. For example, a developer is making changes in a document. Then he opens a browser to read some emails he just received. By opening this browser window, the focus of Eclipse is lost. This could mean that the developer has dropped its goal of changing the document. While this is true for this example, what if he opens a browser to quickly look at some documentation about the library he is using, or to copy a snippet of code? The goal of changing the document is not lost in this case. Therefore, the focus and unfocus of workspace window events are not used.

**Opening/closing of parts** This event fires when a part (the window containing the contents of a document) [16] is opened or closed. Opening a part does not mean that the developer is about to write any code. The developer might just have opened the part to read from it instead. However, closing a part indicates the end of a writing interval. When a developer has edited a document and then decides to save and close that document, obviously the goal of editing that document is dropped. Therefore, the event of closing a part is used to close an existing writing interval for that specific document.

**Changes in a document** This event fires when the content of a document is changed. A change is anything that makes a difference in a document, including whitespaces. Editing a document always results in these events getting fired. When a developer starts writing in a document, this event fires and the beginning of the interval is marked. As soon as the developer decides to write in some other document, the goal of editing shifts to the new document. Now the interval is closed and a new interval is marked for this new document. This would also indicate that an interval is never closed, as long as a developer would not write in some other document or close the document. This is where timeouts come in.

**Writing timeouts** Timeouts make sure that an interval is closed by reason of inactivity. Imagine working on a document for 10 minutes, go for lunch for an hour, then return and close the document. We would like to record an interval of 10 minutes, but without a timeout, we obtain an interval of 70 minutes. This information is incorrect as these 60 extra minutes were not spent on writing code. To solve this issue, we check for changes on a regular basis. As soon as the document has not been changed for a certain time interval, meaning the developer has not altered the code, the interval is closed.

### Reading

We define the 'reading of a document' as having focus on a document in order to understand what is written. A developer can only focus on one document at a time as it is not humanly possible to read more than one document simultaneously. Would the focus change to some other document, the first interval should close and a new one should be created.

Determining when a developer is reading is a lot harder than determining when a developer is writing. Writing means that the document actually changes, while reading does not change the document at all. Even though we can never be sure of when a developer is reading from a screen, there are a few hints that indicate that this is the case. The useful ones considered are described below.

- Opening/closing of parts [16]
- Cursor and scroll bar changes

**Opening/closing of parts** This event fires when a part (the window containing the contents of a document) [16] is opened or closed. When a part is opened, it is assumed that a developer is about to read the content. If some older document was opened before this, this older document automatically closes. This means such an event opens a new interval while closing the other.

**Cursor and scroll bar changes** The knowledge of when a reader is inactive is dependent on the knowledge of a reader still being active, reading the document. However, there is no real way of telling that a reader has walked away from his screen by looking at IDE events. There are only two of these events useful in this situation, namely the cursor change events and the scroll bar movement. A cursor change event is fired when the cursor (not the mouse pointer) inside the document moves from one place to another. Besides this, scroll bar movements are also used to determine a reader being active. Both of these events refresh a timeout.

**Reading timeouts** Comparable with timeouts for writing in a document, we use timeouts for inactivity of reading a document as well. We need these timeouts because we can not tell if a developer has left his desk in order to get a cup of coffee for example. Whenever a developer has been inactive for a certain time period, a timeout event is fired. This will close the open reading interval. Might the developer return to read the same document, a cursor change event or scroll bar movement is required in order to create a new interval for that document.

### Determining timeout values

Determining usable values for timeouts is important. For example, a developer has worked for a minute before getting distracted by a colleague, resulting in ending the activity. Defining the timeouts too large will end up with intervals that are too large compared to how it played out in reality. Making it too small will cause intervals to close too quickly, resulting

in a collection of many very small intervals. We need to give the developer some time to take a moment to think before assuming he is done with his work. Determining a usable timeout value comes down to experimenting. There is no 'right' timeout value, because every developer behaves differently. Therefore, we look for an acceptable default value but give the option to each developer to change this to any other value.

When looking for these values, we must try to meet the following criteria:

- Make the timeout as small as possible to prevent recording intervals that are too large
- Avoid splitting an interval into two intervals, indicating two separate tasks, that in reality was just one task

After experimenting with different values with three students that were willing to cooperate, 15 seconds seemed to be a usable timeout value. No intervals were split while those should have been in the same interval. At the same time, no more than 15 seconds are added to an interval that is closed because of inactivity from the developer. Note that these values can be changed in the preference page of WatchDog.

### 2.2.3 Output

#### XML export

While WatchDog is active, it gathers all recorded intervals in memory. Upon closing Eclipse, they are saved in a serialized form in a folder within the Eclipse installation for later use. At the end of the experiment, the developer has the option to export all gathered intervals by clicking on the WatchDog icon in Eclipse. This will generate an XML file comparable to the example given below.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<intervals>
   <interval>
      <document>
           <projectName>TimeDistributionPlugin</projectName>
           <fileName>DocumentDeActivateEvent.java</fileName>
           <documentType>PRODUCTION</documentType>
      </document>
      <start>1362490233633</start>
      <end>1362490240889</end>
      <duration>7 seconds and 256 milliseconds</duration>
      <activityType>Reading</activityType>
   </interval>

   <interval>
      <document>
           <projectName>WatchDogTest</projectName>
           <fileName>CreateProjectAndClass.java</fileName>
```

```
            <documentType>TEST</documentType>
        </document>
        <start>1362490244085</start>
        <end>1362490249649</end>
        <duration>5 seconds and 564 milliseconds</duration>
        <activityType>Editing</activityType>
    </interval>

    <interval>
        <document>
            <projectName>WatchDogTest</projectName>
            <fileName>CreateProjectAndClass.java</fileName>
            <documentType>TEST</documentType>
        </document>
        <start>1362490240900</start>
        <end>1362490249655</end>
        <duration>8 seconds and 755 milliseconds</duration>
        <activityType>Reading</activityType>
    </interval>
</intervals>
```

This XML output example shows three recorded intervals. The first interval describes the task of reading production code for 7+ seconds. The second interval describes the task of editing a test class for 5+ seconds. The last interval describes reading that same test class for 8+ seconds. Note that these intervals have very short durations as this is merely an example.

### Post-processing

One can imagine that after a month of hard developing work, these XML files can get very big. This makes them useless without proper post-processing. The goal of post-processing is retrieving information from the XML files such that we can answer the following questions:

- How much time did the developer spend on reading production code?
- How much time did the developer spend on writing production code?
- How much time did the developer spend on reading test code?
- How much time did the developer spend on writing test code?

To post-process these XML files, MicroSoft Excel is used. After importing the XML file, the sums of the durations are calculated for each of the four variables listed above. With

| | Production | Test | Totals | Ratios |
|---|---|---|---|---|
| Read | 47164144 | 4157253 | 51321397 | 0,418457895 |
| Write | 62204088 | 9118627 | 71322715 | 0,581542105 |
| Totals | 109368232 | 13275880 | 122644112 | |
| Ratios | 0,891752814 | 0,108247186 | | |

Table 2.3: Example of post-processed XML output

these absolute values representing the amount of milliseconds, a table is created including the ratio between all these four variables. Table 2.3 is an example of such a table.

This example shows that looking at the absolute read and write values does not provide any useful information by itself. The ratios however, give a clear view of what is going on. The 0.89 value in column 'Production' and row 'Ratios' tell us that this developer spend 89% on production code. We can also see a writing percentage of 58%.

Using this table alone is helpful, but since the developers are being interviewed and confronted with their own data, a visual representation is of great value. A graph is created using the data in the table. The example on figure 2.1 shows such a graph.

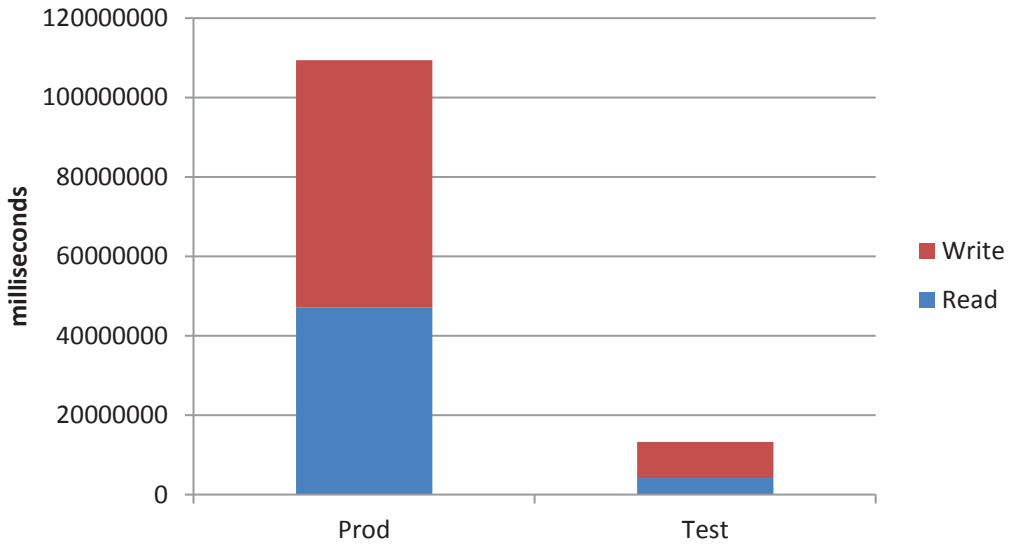


Figure 2.1: Example of time distribution graph

In this graph, the percentages in the table are visualized. The 89% for production code stands out a lot more in this form. Both bars are split up to distinguish writing and reading time.

## 2.3 Validating WatchDog

Before using WatchDog as a tool to do this research, it is important to be sure that the tool is reliable. Without validation of the tool, the results of the experiment could be wrong and therefore useless. This validation process consists of two phases. First, we want to make sure that WatchDog does not cause any problems while developing in Eclipse, the validation of stability. This includes Unhandled Exceptions, crashes and other unexpected behavior that would interrupt the programmer doing his work. The second phase consists of verifying WatchDog's output. The validation in this phase makes sure that WatchDog's behavior, the calculation of intervals, is correct. This is called the validation of behavior.

### Experimental setup

**Validating stability** Besides unit and integration testing during the development phase, WatchDog was installed in multiple different Eclipse installations of five students willing to cooperate. Even though it was believed WatchDog was stable at this phase, 2 out of 5 students were experiencing the same problem. When a user moves a file in Eclipse while an interval was active, a null pointer exception was thrown, resulting in Eclipse showing error messages as pop-ups. After investigating this issue with the use of error logs, this problem was fixed. The same process was repeated with the newer version of WatchDog that did not have this problem. For one month, no issues were reported and WatchDog was declared stable.

**Validating WatchDog's behavior** The goal of validating WatchDog's behavior is to see if it creates the same time profile as a human being would while monitoring a developer. This validation process was done with the help of two students. Each student was given the assignment to continue to develop like they normally would for 45 minutes. During these 45 minutes, their screen was recorded using screen recording software. Besides this recording, we sat down for the whole 45 minutes to make notes of every single non-developing activity. This includes but is not limited to checking their phone, looking for information on a web site or being idle while staring at the screen. With the help of the screen recording and the notes that were made during these 45 minutes, a list of intervals is created manually. The description in table 2.4 below represents one of those intervals:

| File name | Type | Start | End | Duration |
|---|---|---|---|---|
| MyClass.java | Write | 0:03:30 | 0:06:14 | 0:02:44 |

Table 2.4: Example of an interval created manually

This example shows an interval for writing in the file named 'MyClass.java'. The action started 3 minutes and 30 seconds after recording commenced and ended 2 minutes and 44 seconds later. After the whole list of intervals is manually created, the WatchDog XML output is compared to the manual list. Every single interval is checked for differences. If a difference is found, the recorded screen video material is consulted to verify which of the

two lists had the correct information for the interval. Minor differences like durations that are 1 or 2 seconds off, are not taken into account. All other differences are important, these include:

- missing intervals
- intervals that were split up in one list, but not in the other
- intervals with time differences of more than 3 seconds

The more the WatchDog list overlaps with the manually created list, the better WatchDog has recorded the developer time profile. Any mistakes are investigated and fixed before repeating this process.

### Results

The results of this experiment were very positive. The intervals that WatchDog had recorded were very similar to the list of intervals that we created manually. The experiment for the first student provides some interesting results. For 27 intervals that were to compare, only three intervals were different. Two of them were the result of the student checking his phone. This was immediately noticed by us, but WatchDog noticed it 15 seconds later, because of the timeout being triggered. While this is a significant difference, this is expected behavior. The other difference was the result of us missing a small interval of three seconds. This was an error by the observer, not by WatchDog. The second experiment provided roughly the same results, and therefore will not be discussed. These positive results support the correctness of WatchDog. Since no mistakes were found, WatchDog was declared to show expected behavior.

### Shortcomings

Although WatchDog does well in achieving its goals, it is not perfect. First of all, it is not completely accurate in determining development intervals. Timeouts have the possibility to close intervals too soon. This can happen when a developer is staring at his screen, trying to understand a difficult algorithm for example. The timeout event is fired, because no cursor change or scroll bar movement occurs, even though the developer is still reading. Besides wrong interval predictions, WatchDog does not support many configurations. It is only built for Eclipse. Using any other Integrated Development Environment is not supported. Also any other language than Java is not recognized. And, if we want to distinguish testing code from production code, no other testing framework than JUnit 4 [17] must be used.

WatchDog can only record changes to documents. This means that we can not detect if a developer is running test suites or writing GUI tests with Selenium [18] for example. Also running tests as simple as using a console to print messages are not detected. This could potentially give faulty results if these undetectable events happen on a regular basis.

Lastly, WatchDog provides its output as an XML file, filled with many intervals. This data alone is not very useful, post-processing is required and has to be done manually. For future work, WatchDog could be expanded to a plugin that can visualize a developer's time spending on the fly, without the need of post-processing.

# Chapter 3

# Experimental setup

Because WatchDog is thoroughly tested and validated, we can use it to start the experiment to answer the research questions. This experiment consists of multiple phases.

1. The first phase consists of finding a number of companies willing to cooperate in the experiment including monitoring some of their programmers. We managed to find three different companies, initially offering to provide around five different developers to be monitored.

2. Meetings are scheduled with the person responsible in the company for the experiment. In these meetings, student and company decide on when to start the experiment, how long the experiment will run and how much information can be published in this thesis in regard to sensitive information. Besides this, the configurations of IDEs used are checked. This is important for knowing what configurations WatchDog should support.

3. At the time of the decided starting point of the experiment, WatchDog is installed on the machines of the participants. Might any problems arise in this process, this is communicated via e-mail or in person, if necessary, and fixed.

4. At the end of the WatchDog monitoring period, the results are sent to the student in XML format. These results are then post-processed to collect valuable conclusions about the time the developer has spent while programming.

5. The developer is interviewed with questions regarding the period of time while the experiment was active. This interview consists of three parts with a total of 14 questions and will take about 30 minutes per participant. The full interview can be viewed in section 3.3.

6. All interview information is gathered to try and get a global conclusion of that company, if possible. These conclusions per company are used to answer the research questions.

## 3.1 Context

Three different companies decided to participate in this experiment: TOPdesk [11] in Delft, the Software Improvement Group [12] in Amsterdam and InfoSupport [13] in Utrecht. Besides these, a master student of the Technical University in Delft working on the open source CrawlJax [19] also joins the experiment.

TOPdesk develops service management solutions for every type of organisation. The company has multiple offices around Europe with their headquarters located in Delft, the Netherlands. TOPdesk has between 450 and 500 employees. The experiment was held at the office in Delft.

The Software Improvement Group (SIG), located in Amsterdam, provides insight and analysis into the quality of software systems of any organization. They help gaining control of IT landscapes [12]. SIG has built software to measure the quality of software systems by analyzing source code. Each system that is analyzed gets a star rating (1 - 5) based on quality compared to all other customers. The experiment was held at the office that builds this analyzing tool.

InfoSupport provides software solutions, maintenance, hosting and trainings. They have multiple offices around the Netherlands and Belgium with 300+ active employees. The experiment was held at the office where the software for scheduling timetables for the train company NS [20] is built.

The master student works on the CrawlJax [19] project. CrawlJax is a web application crawler that uses the browser to crawl a website or web application by doing a lot more than a regular crawler. Most regular crawlers just look at links to other pages, missing any AJAX calls or other javascript functionality. CrawlJax mimics a real user, which results in finding a lot more states in a website or web application. This information can then be used for testing for example.

For privacy reasons, the results for the three companies will be anonymous. Therefore, the companies will now be referred to as ‘Company A’, ‘Company B’ and ‘Company C’ in no particular order.

## 3.2 Watchdog installation

The Eclipse [8] plugin to record time distribution, WatchDog, can be easily installed using Eclipse by accessing the WatchDog update site [21]. The installation did not cause any problems to the Eclipse installation of any of the developers. However, one company did not allow altering of the Eclipse installation. To solve this problem, a custom eclipse installation was created where WatchDog was installed to eliminate this problem. This installation was used instead for the total time of the experiment to make participation possible.

## 3.3 Interview

The interview for each developer has the goal to explain the WatchDog results and the differences between these results and the developer’s own time spending image. Each in-

terview takes about 30 minutes to complete. The interview consists of three different parts. The first part asks the participant for information regarding the company's testing policy. This also includes any rules that developers have to follow. Besides company rules, we are interested if tools that aid developers with improving software quality are used and whether these are mandatory. Lastly we ask for a developer's own motivation to spend time on test code. These answers can help explain the results that WatchDog has recorded.

The second part of the interview aims to let the participant think about how much time he or she spent on what kind of activities. These are rough estimates and are not meant to be 100% precise. The developer often forgot about some activities of a one-month work period next to the fact that it is hard to give exact numbers when asking for how one has spent their time in one month. Lastly, we ask if test driven development [10] is used. After this part, the post-processed WatchDog results are shown to the interviewee.

The last part aims to create a discussion between the interviewer and the interviewee about the differences between the WatchDog results and the answers given in the second part of the interview. This discussion can lead to interesting conclusions concerning a possible misformed view of a developer.

The next listing shows the interview used.

**Creating image of company regarding testing**

- How would you summarize the testing process of the projects you have been working on? (Which kind of tests, when to test)
- How well would you say you follow these rules? (Scale 1-5, with 5 being 'very well')
- Within your projects, do you have a policy regarding (codified) tests? Which testing tools do you use (testing framework, coverage measurements, mutation testing, lint-style code checkers (static analysis), TDD)?
- What is your motivation for writing tests? Do you write test for checking correctness? For documentation? For building confidence in your code?
- Which interesting events during the experiment did take place and when?

**Time spending estimation**

- How much time do you think you spend on writing tests relative to the amount of time writing production code?
- How much time do you think you spend on reading code relative to actually writing code?
- How much of the unit tests and production code were you involved with and did you write yourself?

- Do you spend time reading test code to write production code or vice versa? (Is the developer using a test driven development approach?)

**Result evaluation**

**Test vs production**

- Can you explain the ratio in the results?
- (If more production time than expected) Why do you think this difference is present?

**Reading vs writing**

- Can you explain the ratio in the results?
- (If more or less reading time than expected) Why do you think this difference is present?
- (When spent more than 5% of time on testing and there is relatively more reading in production or in testing code) Can you explain this phenomenon?

The parts in the interview that are between parentheses in the last section are conditions. Meaning that those questions will only be asked when the condition is met.

# Chapter 4

# Case Studies

This chapter describes the results that were gained for each company and the open source project. We start by discussing WatchDog's measurements by showing the total amounts of time spent during the experiment. After that, we discuss the outcome of the interviews. We combine these results with a summary. We end this chapter by doing an exploratory study on the recorded data over time.

## 4.1 Company A

For company A, six developers were participating in this experiment. These developers willing to cooperate were all in different Scrum [22] teams working on different projects. Their tasks mainly consisted of writing new features and maintaining existing (legacy) code. None of them were designated testers.

### 4.1.1 Watchdog results

The values that were recorded by the WatchDog plugin show interesting results as there was very little time spent on writing and reading test code. The distribution of the time spending percentages of test code is displayed in figure 4.1.

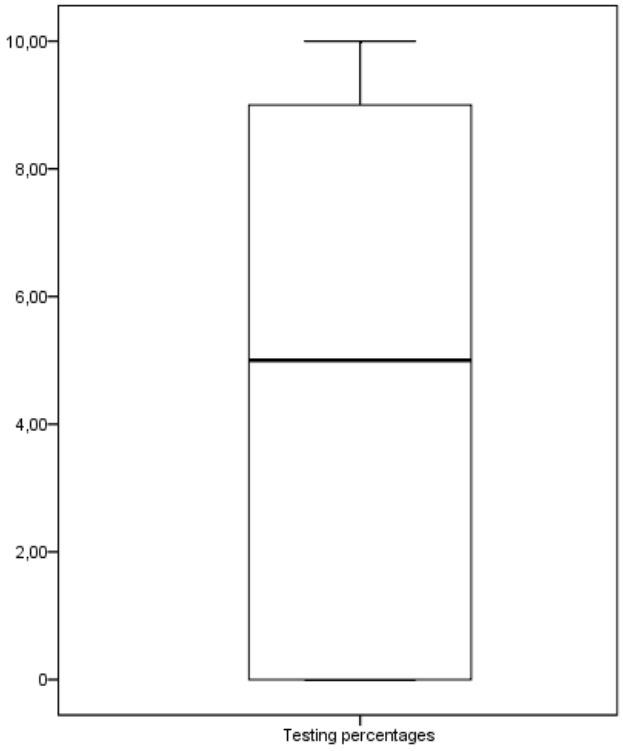


Figure 4.1: Boxplot of testing percentages for Company A

The highest percentage measured is 10%. The lowest value measured is 0%, a value that was measured three times out of the six developers. The most common testing percentages are measured between 0% and 9%.

The values measured for time spending on reading code compared to writing code is a more spread-out distribution. This can be seen in figure 4.2.

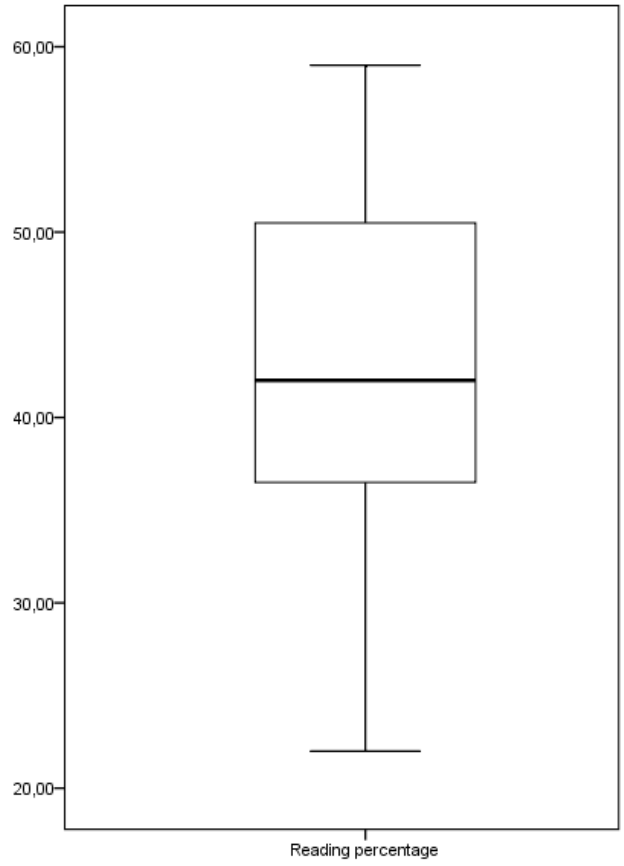


Figure 4.2: Boxplot of reading percentages for Company A

The highest percentage on reading code measured is 59%, while the lowest value measured was just 22%. The most common reading percentages are measured between 36% and 50%.

### 4.1.2 Interview results

#### Company image regarding testing

The low amount of time spent on testing can be explained by the testing policy of the company. The rules of when and how to test the code that a developer writes, are vague. Also, developers mention that their supervisor would not disapprove the lack of writing automated unit tests. This also explains why, when given the question of to rank themselves on following testing policy rules, developers rank themselves mostly 4 out of 5. When the rules are vague, it is easy to follow them.

Tools like static analysis or code coverage measuring systems were not mandatory but are used if the developer likes to use them. Two out of six developers state to use these tools while developing. When asked for the main reason a developer spends time on test code, every developer had their own motive. The three most frequently given answers are as follows:

- For documentation
- For checking correctness (behavior)
- For building confidence in engineered code, so it can be committed without worries

#### Time spending evaluation

When asked for an estimate on how much a developer thinks he or she had spent time on test code versus production code, the estimates were very accurate. The developers that had spent very little time on testing all provided an accurate estimate. The accuracy is decreasing with the time spent on testing increasing. Figure 4.3 shows the estimation of time spent on test code compared to what was measured for each developer. Note that both developer 1 and 3 provided 0% for estimating time spent on test code.

The biggest difference between the estimate given and the measured value is 9%. Explanations of why the estimate is larger than the measured value all follow the same trend. They blame the code for being untestable. This is true for both recent code and legacy code. Fixing this problem of untestable code is too much of a time-investment at that moment. Besides the low quality of code, the developers state to have worked with a lot of GUI code, which is not tested with JUnit [17] or not tested at all. A far less popular answer is the existence of a designated tester which was mentioned by just one developer out of all six participants.

The estimates for how much a developer has spent time reading code versus writing code were less accurate. Four out of six developers thought they spent more time on reading than was actually the case. The differences in the estimates and actual measured values can be seen in figure 4.4.

The most popular reason given for a higher reading expectation, is the fact that a developer knows a lot more about the system they are working on than they initially thought. Also, when working with legacy code, the code is well-documented, albeit not in the form of unit tests. A bit less popular reason is the fact that when new code is written, one needs less time reading up on otherwise code that needs to be maintained.

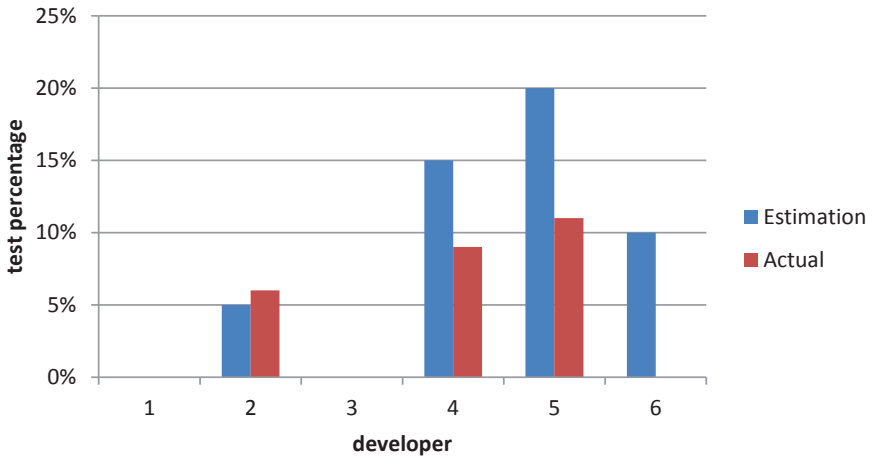


Figure 4.3: Estimations compared with measured values for time spent on test code for company A

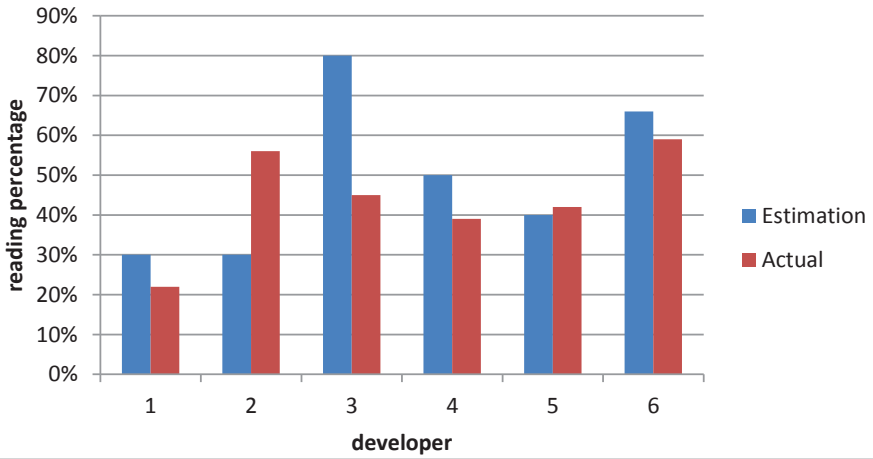


Figure 4.4: Estimations compared with measured values for time spent on reading code for company A

### 4.1.3 Summary

The low amount of time spent on reading and writing test code can be explained by the very loose rules regarding testing software. For the developers, there are no direct consequences for not writing a sufficient amount of unit tests for production code that is being worked on. The reason given by the developers is the fact that the code is of bad quality and therefore untestable. This could be fixed by increasing the quality of existing code to make it more testable. However, developers never feel like actually doing that because they value new features more compared to improving old code.

While developers state the code they work on is of bad quality and untestable, we could expect to see quite a high reading percentage. That is only true in case of altering existing code, not writing new code. This was not the case however with reading values between 36% and 50% , meaning that more time was spent on writing than reading code. So even though the code was of bad quality, the knowledge to alter existing code was there. This was obtained by documentation or memorization.

## 4.2 Company B

For company B, one developer participates in the experiment. Originally two developers were supposed to participate but sadly for one developer the WatchDog data was lost and could not be recovered. The developer that remained works in a scrum team with sprints of a 2 week duration. WatchDog was active for two complete sprints. Besides being a developer, the participant is also the scrum master.

### 4.2.1 Watchdog results

After a period of 4 weeks, WatchDog has measured a very high amount of time spent on reading and writing test code. 67% of time spent was put in reading and writing test code relative to the 33% of time spent put in reading and writing production code. For both test code and production code, more time was spent on reading compared to writing. The reading percentages on the two different activities are quite similar with 63% reading time on production code and 60% reading time on test code. For both activities combined, 61% of time was spent on reading while writing was measured with a percentage of 39%. See table 4.1 and figure 4.5.

| | Reading | Writing |
|---|---|---|
| Production | 63% | 37% |
| Test | 60% | 40% |

Table 4.1: Percentages for company B

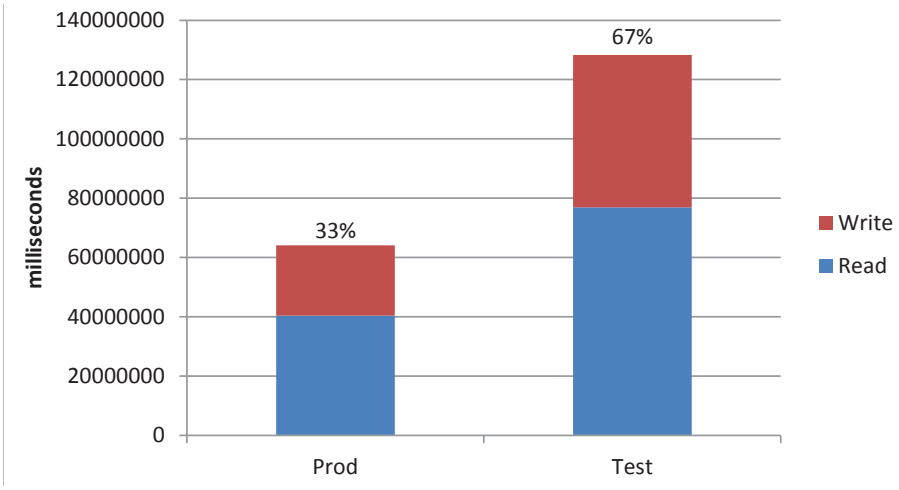


Figure 4.5: Measured values of developer company B

### 4.2.2 Interview results

#### Company image regarding testing

The most remarkable WatchDog result is the high percentage dedicated to the reading and writing of test code. The interview made clear that developers follow strict rules concerning testing their software. The most important ones are listed.

- Every class that has any kind of logic in it should be tested, and aimed to test all branches.
- The developers write test classes themselves, there are no designated testers that write unit tests.
- They try to work with TDD but this does not always happen. However, while watchdog was active, TDD was used.
- At all times, unit tests and production code go hand in hand, tests are never written afterwards.

The interviewee takes these rules seriously and rates himself a 4 out of 5 when asked for how well the rules are followed.

Multiple tools are used to increase software quality and are mandatory for each developer. Examples of tools used are Findbugs [23] and Eclemma [24]. Tools like Findbugs are used in nightly builds. Whenever these tools detect errors in committed code, the developer's code is rejected and fixes are required until no errors are left.

For the developer, the main reason to spend time on maintaining test suites is for refactoring. It is important to know when certain parts of software behave differently when altering existing code.

According to the developer, there was a significant amount of time spent on XML test data, this is not recorded as testing time but does belong to time spent on test code. However, this XML data was not found in the information gathered by WatchDog. It was probably written outside of Eclipse.

#### Time spending evaluation

When asked for an estimation for the production code versus test code ratio, the participant provided a 50-50 answer. This answer would include the missing XML test data which was not recorded. WatchDog however, measured a lot more time spent on test code, a value of 67%. This percentage does not even include the XML test data time, meaning that if it did, the percentage would be even higher. As an explanation for this difference of 17%, the developer said that there were a lot of test suites that needed to be rebuilt, taking more time than initially thought.

The estimation given on how much time was spent on reading code versus writing code was more accurate. The developer provided an estimate of 70% reading time with WatchDog measuring a value of 61% reading time. These values show a relatively low amount of time spent on writing code. This can imply a lack of knowledge of the system being worked

on. In this case, the developer needed to change a lot of existing code, most of which he wrote himself. However, the code was written 2 years in the past, resulting in forgetting the structure of the code for the most part. Therefore, before altering the code and their test cases, more knowledge was required. This knowledge was gathered by reading up on the code.

### 4.2.3 Summary

The high amount of time spent on testing probably has to do with the strict rules regarding testing the software. The rather high amount of time spent on reading code has to do with lack of knowledge of production and test code that is being worked on. It is, however, hard to conclude this because of the lack of data available. Having more data showing the same trends would greatly increase the validity of this conclusion.

The amount of time spent on reading was a lot higher than the amount of time spent on writing. In this case, the developer was working a lot with existing code that needed refactoring. Even though written by the same developer, the knowledge of how the code was structured was lost over two years and therefore time was needed to study the code before altering it.

## 4.3 Company C

For company C, just a single developer was able to participate in the experiment. This is because of the low amount of development work for this company at the time of this graduation project.

### 4.3.1 Watchdog results

After 6 weeks of monitoring the developer, 81% of time spent was put in to reading and writing production code, resulting in 19% dedicated to test code. The difference between reading and writing was very small with a reading percentage of 49% compared to a writing percentage of 51%. Even though this seems to indicate an almost even distribution between reading and writing code for both production and test code, this is not the case. By separating production and test code categories for these values, there is a notable difference. See table 4.2 and figure 4.6.

| | Reading | Writing |
|---|---|---|
| Production | 51% | 49% |
| Test | 41% | 59% |

Table 4.2: Table of percentages for company C

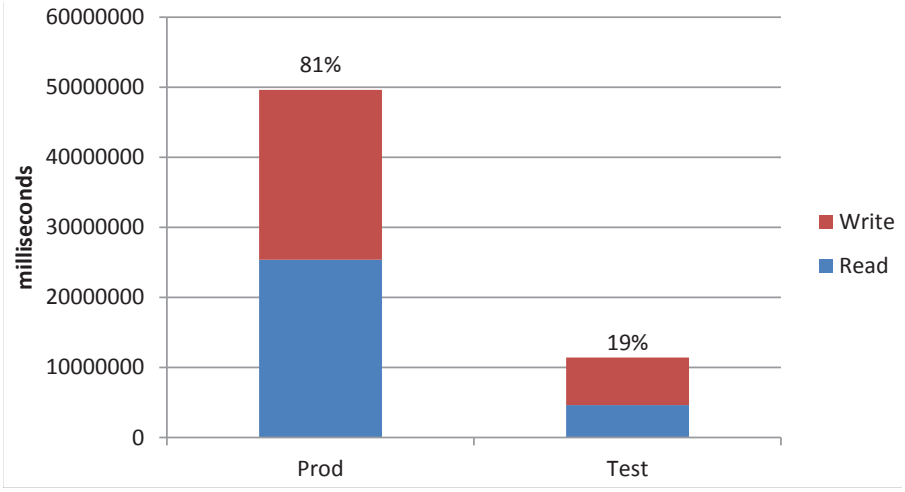


Figure 4.6: Measured values of developer company C

As can be seen from the table, the difference between reading and writing for production code is very small. However, the difference for the same variables when looking at time spent on test code is much bigger. In this case, the developer spent less time on reading test code than writing it. These values combined could indicate that a developer is reading production code to write unit tests instead of the other way around, which happens in Test Driven Development [10].

### 4.3.2 Interview results

#### Company image regarding testing

To ensure a rich test suite for their software, the company follows strict rules with regard to writing tests. Developers have to use Test Driven Development whenever possible. Unit tests should be written for all classes that contain any kind of logic while Selenium [18] is used for testing graphical user interfaces. The developer that was interviewed rates himself a rating of 4 out of 5 when asked how well these rules are followed.

The use of tools is mandatory. Cobertura [25], a test coverage tool, is used to help developers cover all branches when writing a unit test. Total coverage should be between 70 to 80 percent. Aiming for a coverage percentage of 100% is not desirable as trivial methods like getters and setters do not require unit testing. Checkstyle [26], a static analysis tool to detect code smells, is also mandatory. When developers violate the rules set up for Checkstyle, they are not able to commit their code.

Several reasons were given when asked for the main personal reason the developer writes test code. The list below is ordered by importance, starting with the most important reason.

1. Documentation, unit tests are always up-to-date while written documentation can become outdated and filled with false information.

2. Regression testing, having the knowledge of something that was broken by altering code.

3. A guarantee that your code is behaving like it should. Compilation is not enough.

4. Eases debugging by first creating a failing test when a bug is found. Fixing the bug should let the test pass.

#### Time spending evaluation

The participant in this interview was inaccurate about his time spending distribution compared to WatchDog's measurements. 30% time spent on test code was estimated compared to the 19% that was measured. This difference can be explained by the fact that the developer spent more time on refactoring existing production code than initially thought.

The estimation given of reading code compared to writing code is 80%. WatchDog provided a very different value of 51%. This means the developer had the illusion to have spent way more time in reading code than writing it, while this was not the case. An

explanation for this difference was not provided however. The participant had no idea why his estimation was 29% off.

### 4.3.3 Summary

The decent amount of time spent on testing might be the cause of the strict rules regarding testing. Not fulfilling these rules denies the developers of committing their code. Another reason for this might be the fact that the developer values test suites as documentation. Documentation in the form of test suites are always up-to-date compared to written documentation which does not depend on the production code.

The participant estimated a 70:30 ratio in favor of spending time on production code. The measured ratio is 81:19 in favor of spending time on production code. The developer explained this difference with the fact that more time was needed for refactoring existing production code.

The estimate given for time spent on reading was very inaccurate. An explanation for this could not be obtained in this interview as the developer had no idea why time spent on reading was 29% less than estimated.

## 4.4 Open source project, CrawlJax developer

For the last participating entity in the experiment, one developer was monitored for over seven months, starting on the 17th of August 2013 and ending on the 25th of March 2013. Because this developer has worked on multiple different projects, the WatchDog results were first filtered to end up with just the intervals of the project we wanted to investigate, CrawlJax [19]. This project did not use the scrum methodology as the developer mostly worked on CrawlJax on his own and in his free time.

### 4.4.1 WatchDog results

WatchDog measured a time spending value of 29.5% on reading and writing test code relative to reading and writing production code. In total, 45% of time measured is spent on reading code. The percentages for reading production code and test code are 46% and 41% respectively. See table 4.3 and figure 4.7.

| | Reading | Writing |
|---|---|---|
| Production | 46% | 54% |
| Test | 41% | 59% |

Table 4.3: Percentages for the open source project

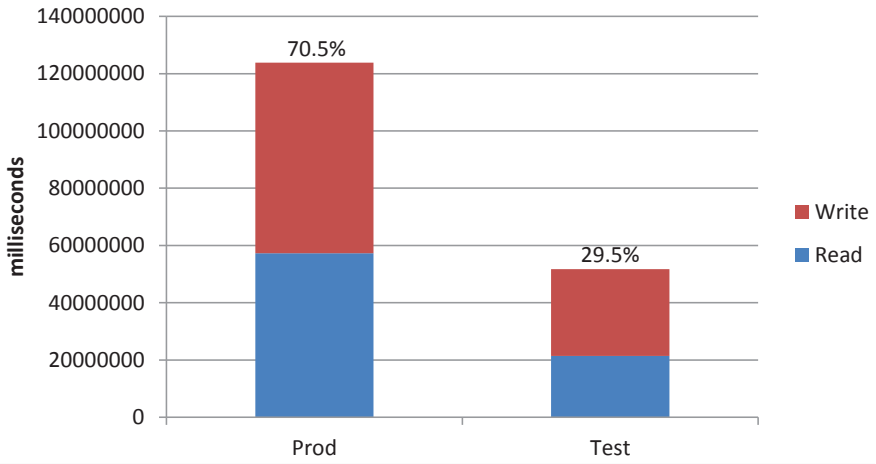


Figure 4.7: Measured values of open source developer

### 4.4.2 Interview results

#### Image regarding testing

The developer that participated in this experiment mostly worked on his own and not in a team. This means that the developer has to make his own rules to ensure his software to be of high quality. The most important rules that were established are as follows:

- All branches should be at least tested by integration tests. Whenever possible, create unit tests as well.
- The GUI is tested by using Selenium [18] tests.
- Test Driven Development is only used when fixing bugs by first creating the test (that should fail) that will succeed when the bug is fixed.

When asked to rate his own obedience to his own rules, a rating of 4.5 out of 5 was given. The developer does not see testing as a burden.

The developer uses multiple tools in his application development. Sonar [27] is an all in one analysis tool that provides information about code coverage, coding rules, potential bugs and more. Mockito [28] provides a mocking functionality to create cleaner and more isolated unit tests.

The main reason for the developer to write unit tests is regression testing. Having these tests, the developer feels confident when altering existing code or adding new features. A second reason is to check the behavior of the code, ensuring the functionality of methods to be correct.

#### Time spending evaluation

The interviewee was very accurate to guessing how much time was spent on reading and writing test code. An estimation of 30% was given, compared to the 29.5% that was measured. The developer states to be very experienced in writing unit tests. In combination with the help of tools, the process of maintaining test suites is not very time consuming. Also, the system itself is not very complex having a small amount of dependencies and subsystems. If this would not be the case, the developer thinks the time spent on reading and writing test code would greatly increase.

While the accuracy on estimating time spent on test code is extremely high, the estimation on reading code versus writing code was off by 40%. A value of 85% of reading time was given compared to the measured 45%. This big of a different is explained by the fact that the developer uses the boy scout rule [29]. This rules defines that any code that you encounter, should be improved when possible. Bad quality code becomes good quality code when inspecting existing classes. This means that reading code results in altering it as well, making it include a writing interval as well. This principle was overlooked, resulting in this 40% difference.

### 4.4.3 Summary

Although this developer works on an open source project without any influence of a company, rules on ensuring software quality are established and followed. The developer values his tools to help analyze source code and make creating unit tests easier. The developer does not see writing tests as a burden.

Estimations of time spent on test code versus production code are very accurate. The developer has a lot of experience with unit tests, making the creation of them easy and therefore more fun to do.

The estimation of reading code versus writing code was way off. The most important reason for this is the boy scout rule that was not taken into consideration when asked for the estimation. With the boy scout rule, a developer that is reading code will often modify it as well. Code that is inspected should always be improved when possible.

## 4.5 Exploring time distribution over time

In this section, we explore the recorded data in more detail. By clustering the activities per day or even per hour, it is possible to create an image that represents a developer's activity over time. By doing this, we hope to find patterns and discuss the possible causes for these to appear. This is an exploratory study. The results discussed in this section are not used in the interview with the developers. Therefore, the findings are not able to be verified.

### 4.5.1 Company A

All participants from company A were measured with a small amount of time spent on testing. The graph in figure 4.8 shows time spending clustered per day for the participant with the highest testing percentage (10%) in this company.

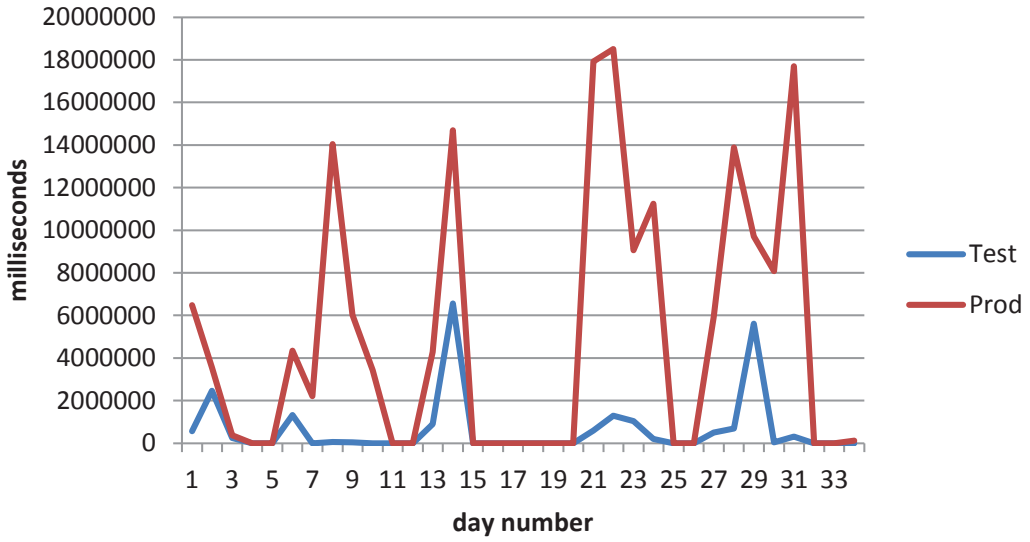


Figure 4.8: Time distribution clustered per day for developer of company A

As can be seen from the graph, the amount of time spent on test code each day never surpasses the time spent on production code. This means that whenever this developer was working with test code, on that same day he or she was always spending more time on production code in total. There is no day in which the developer only focused on test code.

The gap between the 15th and the 20th day can be explained by the fact that this developer was visiting a conference, therefore no activity was measured.

### 4.5.2 Company B

The developer from company B was measured with a 67% of time spent on testing. During the period of 4 weeks that the experiment lasted, whenever time is spent on test code, production code does not stay untouched. The graph in figure 4.9 this by clustering the values per day.

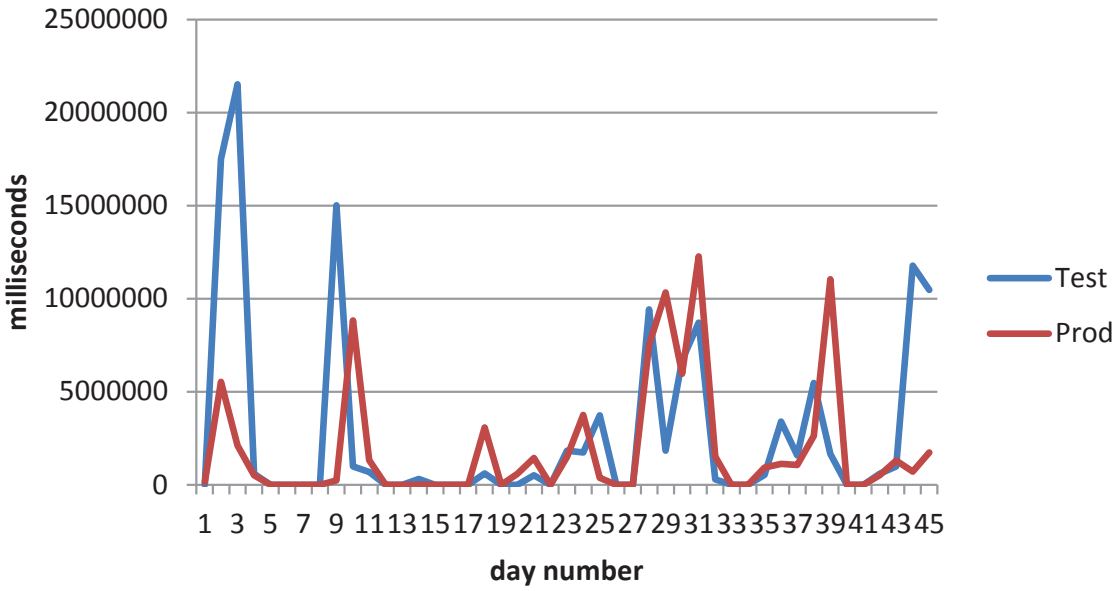


Figure 4.9: Time distribution clustered per day for developer of company B

As can be seen from the graph, the blue line representing time spent on test code, is always accompanied by the red line representing time spent on production code. In this case, these two activities go hand in hand during application development.

This data can also be used to investigate the distribution on a hourly bases, by clustering the timing values per hour. In order to create a useful graph, we will only present a few days. The graph in figure 4.10 is a visualization of how time was spent per hour from the 28th day until the 31st day of the experiment.

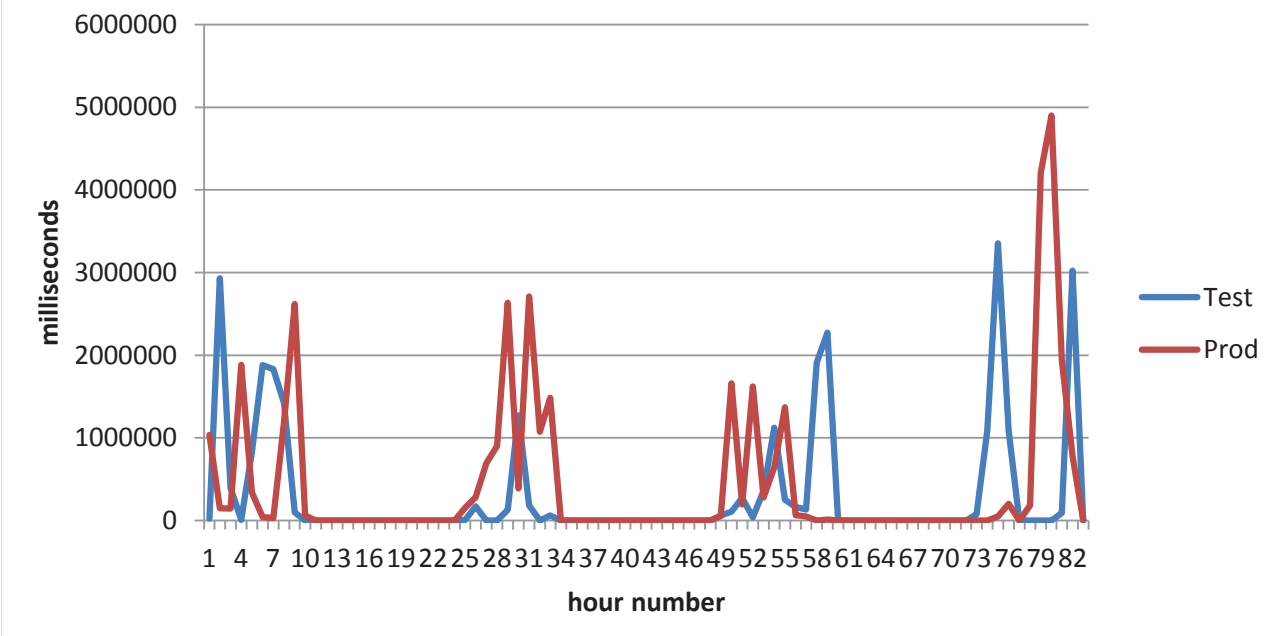


Figure 4.10: Time distribution clustered per hour from the 28th day until the 31st day for developer of company B

In the graph we can clearly see four areas of activity. Each area is a full day of work. From the graph, we can see that the first day consists of multiple phases. It starts with production code and no test code. This is then switched to a phase where there is a lot of time spent on test and just a little time on production code. This process repeats itself until the end of the day. This could be a pattern seen in test driven development. The second day shows different behavior. First of all, we notice that more time is spent on production code here. Only in the middle of this day, more time is being spent on test code. The third day starts with production code but this value decreases as the day goes by. For test code this is the exact opposite. At the beginning of the day, almost no time is spent on test code. As the day progresses, this time increases. This pattern looks like a developer writing tests afterwards, not doing test driven development. Day four clearly consists out of three phases. It starts with a lot of time spent on test code, followed by a lot of time spent on production code. At the end of this day, time is spent on test code again.

### 4.5.3 Company C

The developer from company C was measured with a value of 19% regarding time spent on test code. The graph in figure 4.11 shows that mostly, whenever time was spent on production code, time was also spent on test code.

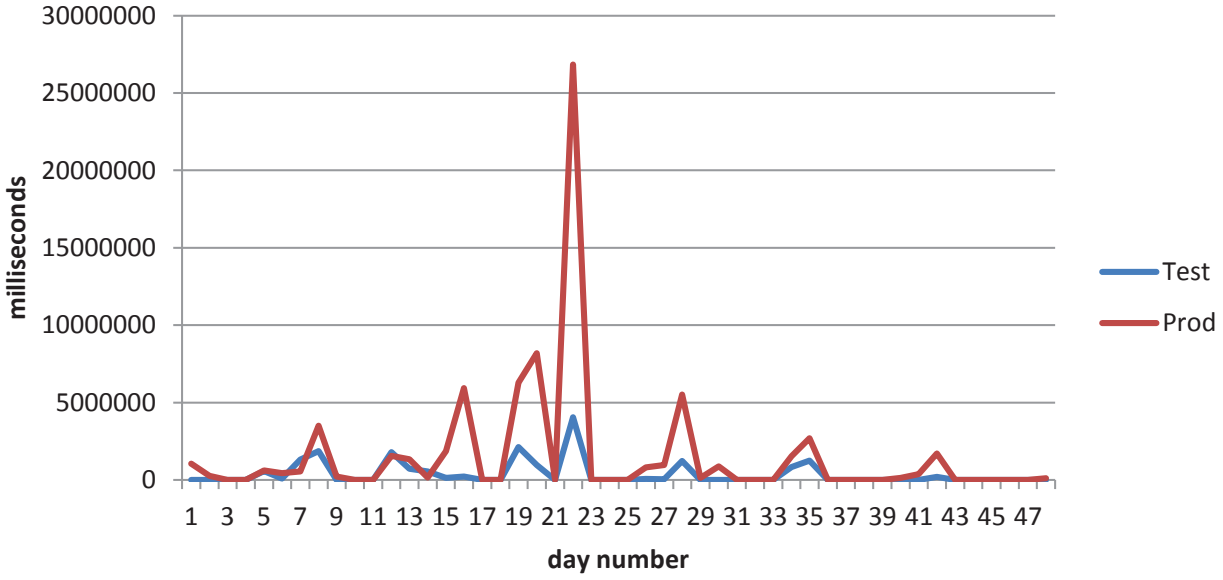


Figure 4.11: Time distribution clustered per day from for developer of company C

The most notable values were measured on day 22 where we detect a large amount of time spent on production code. This could be the cause of an approaching deadline. However, since we did not include the results of this study in the interview, we can not be sure of the cause. The graph on figure 4.12 shows time distribution clustered per hour for four days, from the 18th day until the 22nd day.

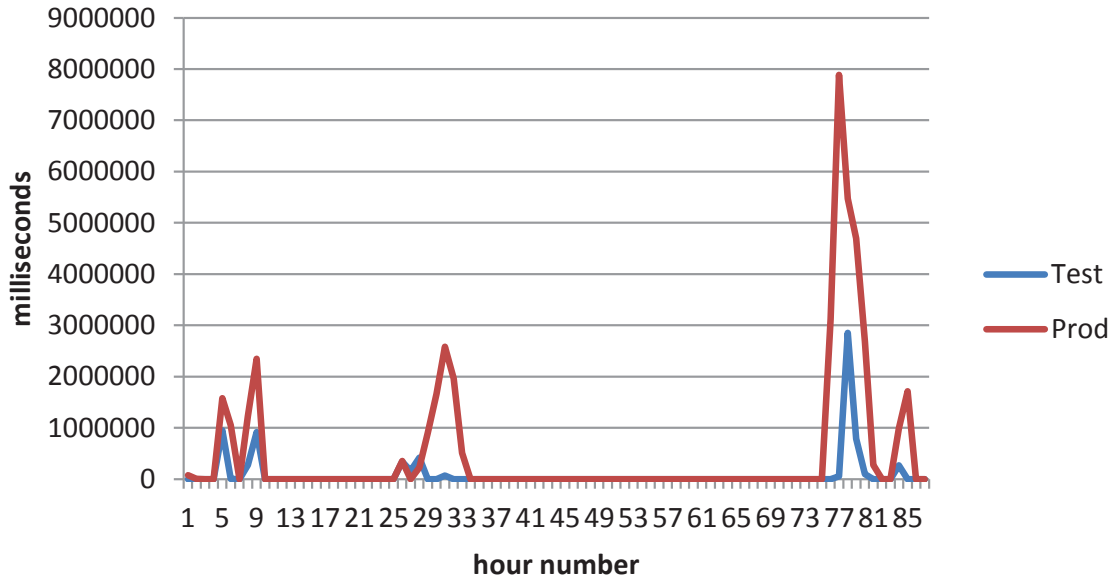


Figure 4.12: Time distribution clustered per hour from the 18th until the 22nd day for developer of company C

First thing to notice is the big gap between the second and third peak. This is the cause of one day with no activity, which can also be seen in the graph with daily values as the 21st day. The first day consists of both test and production code being worked on simultaneously, implying that the developer is not neglecting test code while creating the production code. By closer investigation of the data WatchDog had recorded, we can indeed conclude that the developer was switching back and forth between test and production classes. This could be an indication of the use of test driven development. The second day, compared to the first day, does show just a very little amount of test code activity. The last day is comparable with the first day, albeit with a lot more time spent on both test and production code.

### 4.5.4 CrawlJax developer

The CrawlJax [19] developer was measured with a value of 29.5% regarding time spent on test code. Because this developer was monitored for seven months, creating a graph for the whole period of time would result in a graph that is too cluttered. Therefore, we took a subset of data to analyze, with a duration of 46 days. This graph is displayed in figure 4.13.

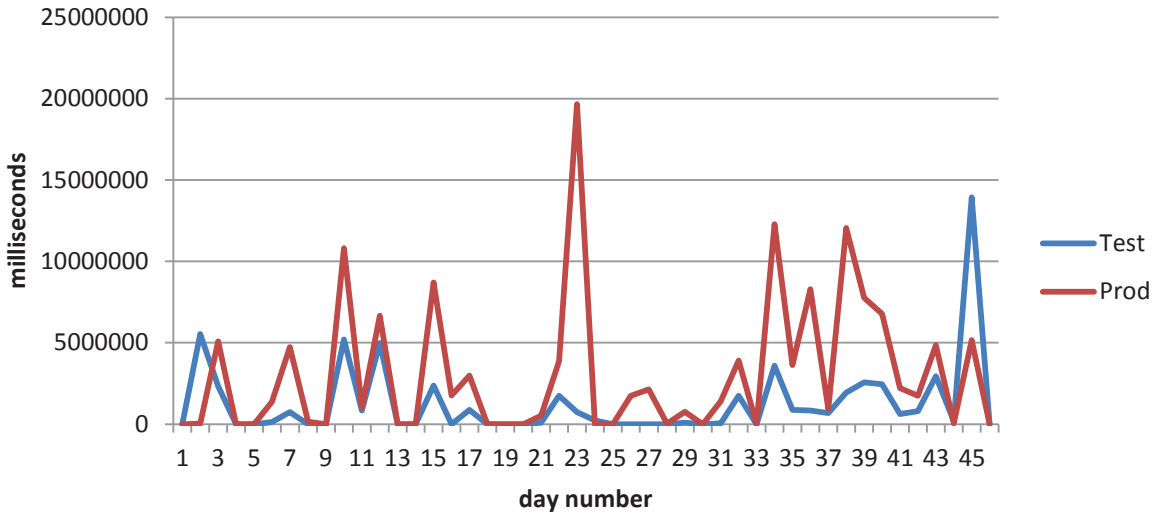


Figure 4.13: Time distribution clustered per day for developer of CrawlJax

This developer is spending time on both production code and test code almost each day. However, on day 23, it seems that test code was neglected. One could expect to see a large amount of time spent on test code one of the following days to maintain the test suites that concern the production code modified on day 23. However, this is not the case.

Around day 39, we see another peak for production code activity while the amount of time spent on test code those days is a lot lower. On day 45 there is a large amount of time spent on test code. A part of this could be time spent on the test code for testing the production code that was modified around day 39. Further investigation of WatchDog's results made clear that indeed two production classes that were frequently modified on day 39, had their corresponding test classes frequently modified on day 45.

The test activity measured from day 9 until day 13 shows roughly the same time spending as the production code activity. To investigate this further, we cluster WatchDog's values by hour in figure 4.14.

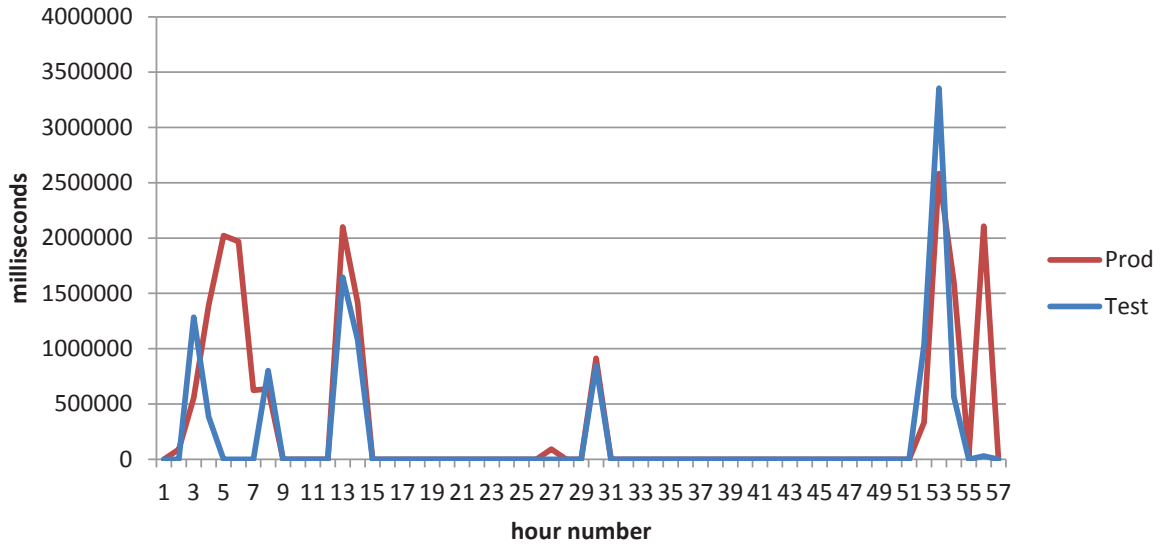


Figure 4.14: Time distribution clustered per hour from the 9th until the 13th day for developer of CrawlJax

This graph shows four areas of activity. In the first seven hours, it seems that the developer is not spending time on both production and test code equally. However, from the 8th hour until the 55th hour, time spending on test and production code are somewhat equal. This could indicate test driven development. By further investigation of WatchDog's results from the 11th until 15th hour, the 29th until the 31st hour and the 51st hour until the 55th hour, we can conclude the following:

**11th until 15th hour** There is a lot of switching between test and production classes during this interval. Judging from the names of the file, the test code that was touched, were test classes for the production code that was touched. This indeed could be test driven development, but bug fixing with the use of a test case could also be a possibility.

**29th until 31st hour** This interval is comparable with the previous interval. Even the same classes are touched. This looks like completing the task the developer was working on between the 11th and the 15th hour.

**51st until 55th hour** This interval is comparable with the previous two intervals, a lot of switching between test and production classes. However, different files were touched.

### 4.5.5 Concluding remarks

In this section we did an exploratory study on WatchDog's results over time. These results were not used in the interviews with the developers, therefore the assumptions made can not be verified.

From the four different developers we analyzed, we were able to see some interesting patterns. For each developer we noticed that when examining daily measurements, test code activity is almost always accompanied with production code. When looking at the graph with the daily values of the developer of company B, it looks like test code and production code are measured at roughly the same time, possibly implying the constant switching between spending time on test code and spending time on production code. However, when clustering values per hour, we do see that this developer is working in phases instead (the research from Zaidman et al. discovered similar behavior through repository mining [30]). This is in contrast with the developer of the open source project. In the graph for this developer with the values clustered per hour, we see that both test code and production code activity is measured per hour. By investigating WatchDog's XML data, we verified that there was indeed a lot of switching between test and production classes.

With more information gathered from the developer's themselves, we could match certain time patterns with development methods. For example, when working in a test driven development manner, we can expect to see both test code and production code activity to be measured simultaneously.

# Chapter 5

# Related Work

In 1975 Brooks et al. published 'The Mythical Man-Month', a book on software engineering and project management [5]. They state that 50% of a project effort is put into testing a software system. Runeson et al. did research on development processes including test effort with data gained from questionnaires, workshops and interviews [6]. This paper concludes that test automation is an issue and companies want to improve on this. However, legacy code and their documentation form a problem in doing so.

Lanza et al. produced two papers where code metrics are used to visualize software systems. The first paper [31] states that as time progresses, a software system gets increasingly complex. Using metrics to visualize a system is useful to the understanding of the system and helps to detect design problems. The second paper uses code visualization to explain how a system has changed over time [32].

Ellims et al. conducted research on the importance of unit tests compared to other forms of validation like code reviews. They conclude that the perceived costs of unit tests is exaggerated and that the benefits in terms of defect detection are quite high in relation to those costs [7]. Another interesting conclusion made in this paper, is the fact that more errors were found during the design of a unit test than during the execution of the test themselves. Data concerning time spending was extracted by examining lines of code.

Zaidman et al. investigated the co-evolution of production and test code in software systems [30]. In order to gain insight into this evolution, version control mining was used in combination with coverage reports to create three different views. The use of these views are validated and demonstrated on two open source projects and one industrial case by observations about the testing processes.

Athanasiou et al. conducted research on the relation between test code quality and issue handling performance [33]. They combined multiple metrics and calibration with other software systems to determine code quality, comparable to the SIG [12] quality model. They concluded that test code quality is not correlated with the speed of handling an issue. However, higher quality test code does increase the amount of resolved issues per month divided by the amount of developers, called the productivity. It also increases the throughput which is the amount of resolved issues per month divided by the number of lines of code.

Robbes et al. introduce the tool 'SpyWare' [34]. This plugin for the Eclipse IDE [8] records changes made in a software system during development. SpyWare's data can be

used to visualize how the software system has changed over time.

Core et al. introduce James [35], a tool for Eclipse [8] that enables developers to write short messages while performing software maintenance tasks. This approach, combined with interaction data from the IDE, makes the knowledge gained during the program comprehension process accessible.

Pinto et al. investigated the evolution of test suites [36]. This paper concludes that 56% of all changes in test suites are additions to the suite in the form of new unit tests. 15% of all changes are test deletions. Out of these deletions, 92% were deleted because of runtime exceptions or assertion errors. The researchers investigated these deletions to see if they were validly deleted or just deleted because a developer was unable to fix them. For the 30 cases that were investigated, all were valid deletions. Another interesting finding is the fact that many valid passing tests were deleted as well. By further investigation it became clear that the removal of these tests did not reduce branch coverage, meaning that the deleted valid tests were probably redundant or moved to some other class.

# Chapter 6

# Conclusions and Future Work

In this chapter we start by giving a summary of this research. After this, the threats to validity are discussed. The third section describes the answers to the research questions. We end this chapter with future work.

## 6.1 Summary

In this master thesis we study the ratio of time spent on production code and test code of developers during application development. In this context, we make the following contributions:

- WatchDog is a plugin created for the Eclipse IDE [8] that transparently records how a developer is spending his or her time as intervals. A collection of these intervals make up a developer time profile, a profile that holds information about how much time was spent on reading and writing production and test code. To our knowledge, this is a new approach as data is recorded in real time, resulting in a more reliable output.

- We used WatchDog to monitor nine developers spread out over three different companies and one open source project to make observations about time spending under various circumstances. We look for correlations between how time is spent and the environment a developer has to work in.

- We interviewed the participants in the experiment to gain knowledge about the environment they have to work in. Also, we confront the developer with WatchDog's results and try to explain them.

Because WatchDog is thoroughly tested and validated, it is able to provide a reliable profile of a developer regarding time spent on production and test code. Since WatchDog is able to distinguish production code from test code, we can create easy to read graphs to not only show the ratio of time spent between production code and test code, but also the ratio of time spent reading and writing code.

WatchDog was installed on the Eclipse IDE of nine developers. Eight out of nine used WatchDog for two sprints, with a two week duration each. The ninth developer used the plugin for seven months. After the WatchDog information was collected, the developers were interviewed to gain insight of how the company treats testing, personal reasons to spend time on testing and how they think they had spent their time while WatchDog was active. After this, the actual results were shown to confront the developers and to start discussions to look for the causes behind the results.

One out of three companies had a low percentage of time spent on test code. For this company, six developers were monitored. The lowest amount of testing time was 0% while the highest percentage was 10%. This could indicate that there are not many unit tests available for the software they were working on. This indeed seemed to be the case, according to interview results. This can be explained by the fact that there are no rules regarding unit tests. Besides this, developers blame the low quality of existing code which makes it untestable. However, there are a lot of integration tests available, not written in Java.

Two out of the three companies, with one developer participating each, spent a lot more time on reading and writing test code with percentages of 19% and even 67%. Interviews made clear that these companies had strict rules regarding unit testing. The following rules are examples of this:

- Every class that contains any kind of logic needs to be unit tested.
- Try to work with Test Driven Development when possible.
- Aim for code coverage of 70% or higher.

Besides these rules, these developers had strong personal reasons to enhance their test suites and did not see maintaining them as a burden.

The last participant that was monitored did not work for a company, but worked on an open source project on his own. Because this is a one-man team, there were no company rules for him to follow. Even so, the percentage of time spent on test code was above average with a percentage of 29.5%. This can be explained by the fact that the developer enjoys maintaining a high quality test suite. He relies on multiple tools that ease the process of writing unit tests such as the mocking framework Mockito and the code analysis tool Sonar [27]. While being very accurate with his own estimation of time spent on test code (a percentage of 30% was given, compared to WatchDog's 29.5%), the estimation for reading code compared to writing code was off by 40% (a percentage of 85% was given, compared to WatchDog's 45%). However, the developer did not take the boy scout rule into account. While the developer did in fact read a lot of existing code, he also improved it along the way. These improvements result in more writing time.

Seven out of nine developers provided an estimation for reading time which was higher than the actual result that WatchDog had measured. Three out of these seven even estimated a percentage 30% to 40% higher than WatchDog's measurements. This implies that most of the participants think they spent more time on reading code than they do in practice. With the information that the interviews have provided, developers state that this error is just a simple estimation error. In other words, developers do not have an accurate image

of how much time it takes to read up on existing code. The developers state that the time spent on reading code could be reduced if that code was of higher quality, had improved documentation and was combined with test suites.

## 6.2 Threats to validity

We describe the factors that may jeopardize the validity of our research in this section. To be consistent, we use the case study guidelines as described in [37] and [38], to divide these factors into four categories.

**Construct validity** Because WatchDog collects data without a human's subjective input, there is no threat to the construct's validity for this part. The questions in the interviews however, could be interpreted differently by the interviewee than what was intended. To make this threat as small as possible, all interview questions are discussed thoroughly when it seems like questions are answered in doubt or the answers lack important information.

**Internal validity** Even though WatchDog's data is reliable, it only supports very specific kinds of activities. For example, only JUnit4+ is supported and all files except for .java files are not recognized. WatchDog can only tell if a developer is working on test code, if they are working in Eclipse, with a .java file with the use of the JUnit framework. Would the developer use XML files to provide data for test cases, WatchDog will not use this information to record time spent on test code. This is a serious threat when developer use a configuration which is not supported. Therefore, before implementing WatchDog, we investigated the configurations of the companies participating.

**External validity** Three companies and one master student were participating in this experiment. In total, nine developers have provided WatchDog data and were interviewed. This number is low. Answers to the research questions are based on this data. In order to provide more reliable conclusions about this topic, more data should be collected.

**Reliability** The observations on the data WatchDog has provided and the results of the interviews are made by the researchers themselves. This means that there is a risk that some other researcher would interpret the data differently. Chances for this to happen are small though as the data is simple and straightforward. The interviews with the participants have a higher risk to be unreliable. Collecting data by means of an interview can be influenced by multiple factors. Some example are given below:

- The mood of the interviewee.
- The clarity of the research questions.
- The urge of an interviewee to answer more positively.

To reduce this risk, we ask the participants to answer the questions in honesty and ask for more clarity when questions are not well-understood.

## 6.3 Conclusions

With the information gathered by WatchDog and the interviews, we can answer the research questions.

**Sub question A:** How much time do developers spend on test code relative to production code?

With WatchDog, we are able to monitor developers in real time to get an accurate view of the ratio of time spent between test code and production code. The values that our tool registered are very dispersed. For the nine developers that are monitored, time spent on reading and writing test code vary from 0% to 67% relative to production code. 0% time spent on testing means that not a single test file was touched. This was the case for three out of nine participants. Our research implies that developers spend less time on creating test suites when the company has not set up any rules regarding unit testing. Another reason for not maintaining test suites is untestable code. Touching this code to make it testable can be dangerous, as there is no way to tell when working code gets damaged, because there are not enough unit tests available. One developer worked on his own, without company rules, but WatchDog measured that he had dedicated 30% of his time on reading and writing test code. This developer does not see unit testing as a burden and enjoys creating test suites with the help of multiple tools to ease this process.

**Sub question B:** How much time do developers spend on reading vs writing code?

WatchDog uses Eclipse user interface events to create a reliable view of the ratio between reading code and writing code. Compared to the percentages of time spent on production and test code, the values for the ratio between reading and writing code are less spread out. The values vary from 22% to 61% of time spent on reading compared to time spent on writing code. From the interview results, we can see a relation between reading time and code quality. Low quality code results in more time needed to read and fully understand it before altering the code. When code needs to be changed that was recently written by the same developer, less reading time is measured. Also, even though it was written by the same developer, code that was not recently written requires time to comprehend fully. Developers state that reading time could be reduced if the quality of the code was higher, was better documented and was accompanied with unit tests.

**Sub question C:** Does the time distribution view of the developer match with the reality?

Eight out of nine developers were very accurate with their estimation of how much time they spent on test code. The biggest difference between their estimation and the measurement of these eight developers was 11%. One developer's estimation was less accurate with a difference of 21%.

Estimating the ratio between reading and writing seemed a lot harder as the majority (seven out of nine) thought they had spend more time on reading than WatchDog had measured. Three developers were off by more than 30% with 40% being the biggest estimation error.

Developers have an easier job estimating the ratio between test code and production code than estimating the ratio between reading and writing code. Most of the developers feel like they had spent more time on reading code than what the measurements provided. This is often explained by the fact that the developer had more knowledge of the code that he or she was working on than initially thought.

## 6.4 Future work

The new way we introduce to do research about how a developer spends his time during application development has a lot of potential. WatchDog as a tool is a step in the right direction to make this kind of research possible. It does however, has its shortcomings. As future work, the following can be improved.

**Support more environments** Enlarge the collection of WatchDog's supported languages, unit test frameworks and IDEs. As of now, WatchDog only supports Java as a language, JUnit4+ as a unit test framework and Eclipse as the IDE. By enhancing this, it is possible to monitor developers working in different development environments.

**Record more data** WatchDog only records changes in documents like opening, closing or editing. For now, we do not record data that does not have to do with documents. For example, running a test suite will go unnoticed. This data could be useful though, as it is a way of spending time with unit tests.

**Automated post-processing** WatchDog provides the output of the recorded data as an XML file. This file contains all intervals with their values. This XML file has to be manually post-processed to extract useful information like graphs or ratios. WatchDog can be improved to do this post-processing work automatically. The plugin could also provide graphs in the IDE itself. However, this should be optional, as it might be desirable to hide the information from the developer for a period of time. This can be important when a developer needs to be interviewed about his time spending without being biased.

**Enlarge the group of participants** For this research, nine developers spread out over three companies and one open source project were monitored. Conclusions based on just these participants are not very reliable. Repeating this research on a larger scale would provide more data, increasing the reliability of conclusions made.

**Combine time measurements with other metrics** Multiple papers have been published about researching software development. For example, Zaidman et al. study the evolution of software systems with the use of version control mining [30]. Athanasiou et al. study

the relations between code quality and issue handling performance [33]. Bruntink et al. study the testability of a software system by examining metrics from the production code [9]. According to this research, multiple metrics are correlated. Lanza et al. use software metrics to study the evolution of classes [32]. One could research the combination of version control mining data, software metrics or other data with time distribution data to see if and how these values are correlated.

**Matching development methods with patterns in time profiles** In this thesis, we did an exploratory study on WatchDog's results over time. However, the developers were not interviewed about this so we were unable to verify certain patterns that emerged when investigating WatchDog's data. As future work, this area of research can be explored to find correlations between a developer's time profile and the development methods this developer states to use.

# Bibliography


[1] http://www.bbc.co.uk/news/magazine-19214294.

[2] Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. pages ES–3.

[3] Todd L. Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel. An empirical study of regression test selection techniques. *ACM Transactions on Software Engineering and Methodology*, 10(2):184–208, April 2001.

[4] David Kung, Jerry Gao, Pei Hsia, Yasufumi Toyoshima, Chris Chen, Young-si Kim, and Young-kee Song. Developing an Software Testing and Maintenance Environment. 38(10), 1995.

[5] Frederick P. Brooks, Jr. *The mythical man-month (anniversary ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[6] Per Runeson, Carina Andersson, and Martin Hst. Test processes in software product evolution - a qualitative survey on the state of practice. *Journal of Software Maintenance and Evolution: Research and Practice*, 15(1):41–59, January 2003.

[7] Michael Ellims, James Bridges, and Darrel C. Ince. The Economics of Unit Testing. *Empirical Software Engineering*, 11(1):5–31, February 2006.

[8] http://eclipse.org.

[9] Magiel Bruntink and Arie van Deursen. An empirical study into class testability. *Journal of Systems and Software*, 79(9):1219–1232, 2006.

[10] http://www.agiledata.org/essays/tdd.html.

[11] http://www.topdesk.com.

[12] http://www.sig.eu/en.

[13] http://www.infosupport.com/.

[14] https://github.com/wouterwillems10/watchdog.

[15] http://c2.com/cgi/wiki?AbstractSyntaxTree.

[16] http://help.eclipse.org/indigo/index.jsp?topic=/org.eclipse.platform.doc.isv/reference/api/org/eclipse/ui/IWorkbenchPart.html.

[17] http://junit.org/.

[18] http://docs.seleniumhq.org/.

[19] http://crawljax.com/.

[20] http://www.ns.nl/.

[21] http://eclipsewatchdogplugin.googlecode.com/svn/trunk/update_site/.

[22] http://www.scrum.org/.

[23] http://findbugs.sourceforge.net/.

[24] http://www.eclemma.org/.

[25] http://ecobertura.johoop.de/.

[26] http://eclipse-cs.sourceforge.net/.

[27] http://www.sonarsource.org/.

[28] https://code.google.com/p/mockito/.

[29] http://www.informit.com/articles/article.aspx?p=1235624&seqNum=6.

[30] Andy Zaidman, Bart Rompaey, Arie Deursen, and Serge Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, 16(3):325–364, September 2010.

[31] Michele Lanza. Combining metrics and graphs for object oriented reverse engineering. *Institut fur Informatik und angewandte Mathematik Universitat Bern*, 1999.

[32] Michelle Lanza and Stphane Ducasse. Understanding software evolution using a combination of software visualization and software metrics. *Lobjet*, 8(1-2):135–149, 2002.

[33] Dimitrios Athanasiou, Ariadi Nugroho Member, Joost Visser Member, and Ieee Computer Society. Test Code Quality and Its Relation to Issue Handling Performance. 2000. Under review.

[34] Romain Robbes and Michele Lanza. Spyware: a change-aware development toolset. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 847–850, New York, NY, USA, 2008. ACM.

[35] A. Guzzi, M. Pinzger, and A. van Deursen. Combining micro-blogging and ide interactions to support developers in their quests. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–5, 2010.

[36] Leandro Sales Pinto, Politecnico Milano, Alessandro Orso, and Georgia Inst. Understanding Myths and Realities of Test-Suite Evolution Categories and Subject Descriptors. 1:1–11.

[37] Yin, r.k.: Case study research: Design and methods, 3 edition. sage publications (2002).

[38] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Softw. Engg.*, 14(2):131–164, April 2009.