On the Effect of Code Quality on Agile Effort Estimations: The Case of Shell

Version of October 11, 2017

Jorden van Breemen

On the Effect of Code Quality on Agile Effort Estimations: The Case of Shell

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Jorden van Breemen
born in Heemskerk, the Netherlands

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

Engineering & Smart Delivery Vertical
Technical and Competitive IT
Projects & Technology
Shell Global Solutions International
Rijswijk, the Netherlands
www.shell.com

On the Effect of Code Quality on Agile Effort Estimations: The Case of Shell

Author: Jorden van Breemen
Student id: 4508696
Email: jvb@bloemsmavanbreemen.nl

Abstract

Agile software development has interested researchers for the last decade. Agile software development teams develop iteration sessions that often last weeks. During development, teams work on technical code and its content. Intuitively, more effort is required to implement new features in poorly constructed code with low quality. This study investigates if and how developers consider the quality of their code during their agile effort estimations. Furthermore, we investigate whether the accuracy of their estimations could increase if developers considered the quality of the code. This study is conducted in a large software development department, that is part of Royal Dutch Shell. We take a mixed method approach, where we interview nine developers and quality experts and mine the repositories of six agile development teams. Initially, we reviewed the existing importance measures of code quality during effort estimations, including how code quality is maintained. We also evaluate the impact of code quality on estimation accuracy.

Developers did not consider code quality high on the priority list during the estimation stage of development. Similarly, we did not find an empirical relationship between the quality metrics and effort estimations. Surprisingly, code quality only had minor effects on the accuracy of the effort estimations. Developers did often encounter quality issues in legacy code. However, overall our study shows that code quality is only of minor importance during agile effort estimations.

Thesis Committee:

Chair:	Prof. dr. ir. Rini van Solingen, Faculty EEMCS, TU Delft
University supervisor:	Dr. Alberto Bacchelli, Faculty EEMCS, TU Delft
Company supervisor:	Dr. ir. Rik Essenius, ESDV, Shell
Committee Member:	Anand A. Sawant, Faculty EEMCS, TU Delft

Preface

This thesis completes the final part of my master's degree study in computer science at Delft Univeristy of Technology. This thesis was done in collaboration with Royal Dutch Shell during an eight-month internship.

Several people deserve my thanks for their support during the completion of this thesis. First of all, I would like to thank my university supervisor, Alberto Bacchelli, for his invaluable guidance. He motivated and advised me in the right direction. Our meetings always led to new insights and ideas. Many thanks to my second supervisor, Anand Sawant, for his continuous support and for always providing me with guidance and motivation. He taught me a lot about academic writing and gave me confidence in my work. I could not have wished for better supervision.

Special thanks to my company supervisors, Rik Essenius and Adam Jordan. I had a great time during my internship and they provided me with all the resources and knowledge required to complete my thesis. They taught me many valuable lessons and provided me with an eye-opening industry perspective.

Thanks to Rini van Solingen, who provided me with valuable feedback that had great impact on my thesis. Last but not least, I would like to thank my family for their love and support throughout my studies.

Jorden van Breemen
Delft, the Netherlands
October 11, 2017

Contents

List of Figures

List of Tables

Chapter 1

Introduction

During the last decade, there has been tremendous interest in agile software development [81]. Agile software development aims at developing software quickly, precise and simply in an environment of rapidly changing requirements [41], in which customer feedback plays an important role [81]. In agile software development planning is done in short iterations, often lasting only a few weeks. The agile planning process is done in three levels: release planning, iteration planning and current day planning [31]. An agile team will plan the iterations by defining a set of goals they need to realize in the iteration. The realization of these goals often require teams to create a set of tasks. To accurately determine the number of tasks and goals a team can realize in the coming iteration, the teams have to estimate the effort required to complete them.

Effort estimation is an integral part of software project management. Effort estimations are often done under the umbrella of the development process [81]. A wide variety of research proposes different models, techniques, methods and tools to estimate and plan software development effort. The way effort estimations are done dependents on the development process the project follows. There are many software development processes besides agile, all having different methods to realize planning [61, 34].

There are several techniques to estimate an agile iteration, such as planning poker [45], expert opinion [32] or analogy and disaggregation [31]. These techniques rely on the experience of the team members, where their opinions define the goals and the estimated effort required to complete these goals. The biggest influence that determines these estimations is often the size of the change [31]. However, iterations in agile software development are short and frequent, so the planning process is done differently from more traditional software project estimations [61].

Development teams will undoubtedly encounter scenarios in which they must modify or maintain existing code. Maintainability in software development is often defined in terms of the ISO25010 standard [74, 17], consisting of the modularity, reusability, analyzability, modifiability and testability of the product [2]. Technical code quality is an important determinant for these characteristics [17]. In agile, quality assurance activities are often integrated into the team [20]. However, to realize these characteristics, effort and commitment from the developer are required.

There are prior studies that focus on the relationship among effort and quality [12, 44,

11]. However, to the best of our knowledge, none of these studies consider importance of code quality for agile effort estimations. Therefore, we hypothesize that, if no time is estimated in the iteration planning for quality maintenance efforts, developers can encounter unexpected quality issues during development. These quality issues can lead to problems in the agile iteration planning, and if not handled accordingly can lead to further quality degradation. This study investigates what effect code quality consideration has on agile effort estimations. Furthermore, we investigate if the accuracy of the effort estimations would have increased if code quality was considered during the estimations.

To this aim, we conduct a mixed method investigation, where we interview nine developers and quality experts. We then mine the repositories of six development teams to find associations among effort, estimations and code quality. These developers and teams originate from the Engineering and Smart Delivery Vertical, a department in the Royal Dutch Shell[1] that is accountable for software applications and their development.

During the interviews, we ask the participants a series of questions about how they define code quality, if how they encounter it during their daily live, and how they incorporate it into their effort estimations. Furthermore, we ask the interviewees how they maintain quality, and if they think it affects their estimations. To further investigate the impact of code quality on effort estimations, we present the interviewees with their development work that suffers from poor code quality and that has peculiar effort characteristics.

To explore the effect of code quality on effort estimations beyond the perspective of the interviewees, we empirically investigate this relationship. Metrics often serve as a good indication of the maintainability of the code [17, 84]. We identify two sets of metrics: code quality and process metrics [69, 62]. We look for relationships among these metrics and effort estimations. We hypothesize that, if developers already think about code quality during their effort estimations, poor quality metrics result in higher effort estimations, while good quality metrics result in lower effort estimations. We evaluate if these metrics could serve as a predictor for estimated effort. The data collection, storage, and visualization are done via a tool we developed to mine and visualize data from Microsoft Visual Studio Team Services [2].

Furthermore, we investigate if considering code quality metrics could impact the accuracy of estimations. We use a method based on user activity to calculate the actual effort spent by developers. We then use this data to calculate the estimation accuracy and analyze if the metrics relate to inaccuracies.

The remainder of this thesis is organized as follows: Chapter two describes the study methodology of the study. Chapter three provides results of the empirical and qualitative research. Chapter four discusses the results. Chapter five suggests related work and Chapter six concludes on the results.

[1] http://www.shell.com/

[2] https://www.visualstudio.com/team-services/

Chapter 2

Methodology

This chapter introduces the research questions and the approaches taken to answer them.

2.1 Research Questions

In this study, we investigate how code quality is associated with agile effort estimations. We hypothesize that, while not the most important driving force of an agile effort estimation [31], code quality is associated with development effort and effort estimations. Research has shown that technical code quality is an important determinant of the ability to maintain code [17]. Hence, we hypothesize that maintaining poor code quality may be linked to effort. Furthermore, if developers do not account for code quality during their agile effort estimations, the accuracy of the estimation could suffer. Therefore, to assess if code quality considerations during agile effort estimations can have an impact on the estimation accuracy, we investigate if there is a relationship among code quality and the accuracy of the effort estimations.

We follow a qualitative and quantitative research method to answer these questions. We interview nine experienced developers and quality specialists in ESDV, a department in Shell. Furthermore, we consider four projects, consisting of a total of six different development teams. We look for empirical ways of measuring relationships between effort and code quality.

We start investigating how developers measure code quality, how they maintain it, and how they consider it during agile effort estimations. The answers will reveal if, why and how developers consider code quality during their estimations. It will inform us about the impact of code quality on developers while maintaining code, and if they consider it important during their agile effort estimations. This yields the following question:

RQ1: How do developers consider code quality during agile effort estimations?

We investigate this question by interviewing developers about how they measure quality and how they incorporate quality into their estimation process. We also want to identify the factors that influence the estimations. Furthermore, we look for a set of tasks that were executed in the past by every interviewee, all of which score poorly in terms of quality

metrics. We ask the developer of the poor quality code if code quality influenced their estimation procedure (Section 2.11).

Next, we investigate the empirical relationship among effort estimations and code quality attributes. While the interviewees might provide valuable insight, we try to further explore the implications of quality metrics during the estimation procedure. We argue that developers may already subconsciously consider quality metrics because they are familiar with the code. Therefore, we formulate the following question:

RQ2: How do code quality metrics relate to agile effort estimations?

We use two groups of metrics: code quality and process metrics (see Section 2.5). The relationships are evaluated by comparing effort estimations to the metrics. We suggest that the developers will estimate higher for code that scores poor on quality metrics. They might also realize that maintaining, and perhaps refactoring, will take longer on poor quality code. Because the relationships between the estimated effort and the quality metrics are likely a combination of a multitude of metrics, we apply two types of multiple linear regression (Section 2.10).

To investigate the impact of code quality on effort, we want to know if, when and how developers encounter code quality issues. Furthermore, we evaluate if these scenarios are considered during agile effort estimations, and evaluate the impact and importance of code quality on the developers' ability to maintain code.

RQ3: When and how do developers encounter code quality during actual effort?

We investigate when and how developers face scenarios that take more effort than estimated to maintain the code. We do so by asking developers a series of questions about their encounters with poor code quality. Furthermore, we present developers with development tasks they recently completed, that are of highly inaccurate and poor code quality. We hypothesize that code quality could have delayed or increased development effort, and hence be the reason for the inaccurate estimation. Furthermore, we want to investigate if developers regularly encounter poor quality to investigate the frequency of poor quality and its impact during the actual effort.

Next, we want to know how accurate agile effort estimations are. This will inform us if there is room for improvement in the agile effort estimations, but also how the actual effort compares to the estimated effort. Therefore, we propose the next research question:

RQ4: How accurate are agile effort estimations?

We define accuracy using the magnitude of relative error [85], a method commonly used to define accuracy in (agile) effort estimations [81]. By calculating the actual time spent on a task (Section 2.9), we can obtain the inaccuracies of effort estimations. Furthermore, we can observe if developers tend to over or under estimate. This method also serves as an assessment of the actual effort calculations.

Finally, we want to investigate if the estimation accuracy could have improved if developers considered code quality metrics. This will help us determine if developers could have benefited from code quality considerations during their agile effort estimations.

RQ5: What is the influence of code quality metrics on the accuracy of agile effort estimations?

To evaluate if considering metrics could have increased accuracy, we look for empirical relationships between the accuracy of the effort estimations and the code quality metrics. We suspect that if code quality is not accounted for during the effort estimations, the accuracy of the estimation might suffer. Hence, we investigate if inaccurate scenarios are associated with poor code quality, but also if good code quality leads to quicker or more accurate estimations. If they do, we would have an indication that considering code quality influences the effort estimations.

2.2 Research Setting

This thesis was conducted in collaboration with Shell Global Solutions, primarily with the Technical and Competitive IT (TaCIT) branch of the Project & Technology department.

2.2.1 Shell and TaCIT

Royal Dutch Shell is one of the *major* oil companies and one of the largest companies in the world [4]. The company is involved with the production, refining, distribution and marketing of petrochemicals, power generation, and oil/gas trading.

The Projects & Technology department focuses on the development and research that leads to innovation and future low-cost investments. Technical and Competitive IT (TaCIT) branch is part of the Projects & Technology division, which aims to create and deploy information technology across Shell.

2.2.2 ESDV and DevOps

One of the three delivery organizations in TaCIT is the Engineering & SMART Delivery Vertical (ESDV). The department is accountable for software systems and applications for project engineering, process engineering, discipline engineering, real time control,and supply chain optimization. In addition, ESDV ensures affordable and competitive life cycle management, from innovation to decommissioning. ESDV employs about three-hundred active staff members.

While ESDV manage and maintain their (software) development projects, not all projects are developed by ESDV staff. Some teams consist of full-time or part-time contractors who are only hired for the duration of the project. Some projects are completely outsourced, and only the project management is done by ESDV.

As of 2017, every development team that produces ESDV software in-house are required to adopt the DevOps way of working. DevOps enhances productivity by increasing

collaboration between the developers/engineers (Dev) and IT operations (Ops) throughout the complete software life-cycle [48, 72, 54].

One of the key components of successful DevOps adoption is the use of an agile work process. Unlike traditional process methods, which frequently advocate extensive planning and a codified process, agile processes rely on the capabilities and creativity of the development team [35]. Teams are self-managed, and frequent collaboration is advocated. Teams adhere to short development cycles, constantly adapting to change and embracing change and customer feedback. Agile processes recognize a variety of implementation types, such as XP, Scrum, Crystal, DSDM, FDD, and Lean [35]. Scrum is one of the most popular implementations, and the method adopted by ESDV.

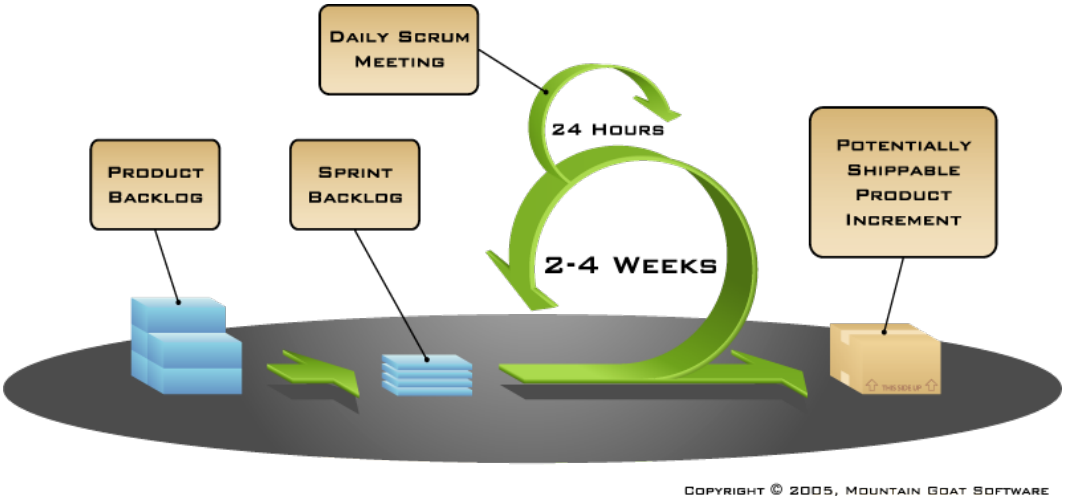


Figure 2.1: The SCRUM method [3]

In Scrum, the team maintains a backlog with a list of features that should be developed to realize the product (Figure 2.1). The teams often plan their iterations in sprints that last two to four weeks. For each sprint, the team selects a set of Product Backlog Items (PBIs) from the backlog to determine the next sprint goal. Teams further divide the goals into individual tasks that are required to complete the sprint. Meetings are held daily to discuss the status of the tasks and debate any impediments.

2.2.3 Micorosft Visual Studio Team Services

ESDV software projects use of Microsoft Visual Studio Team Services (VSTS) [1], a software that functions as a service solution. The platform offers functionalities to help software developers with their builds, release management, test management, version control, continuous integration and work item tracking.

[1] https://www.visualstudio.com/team-services/

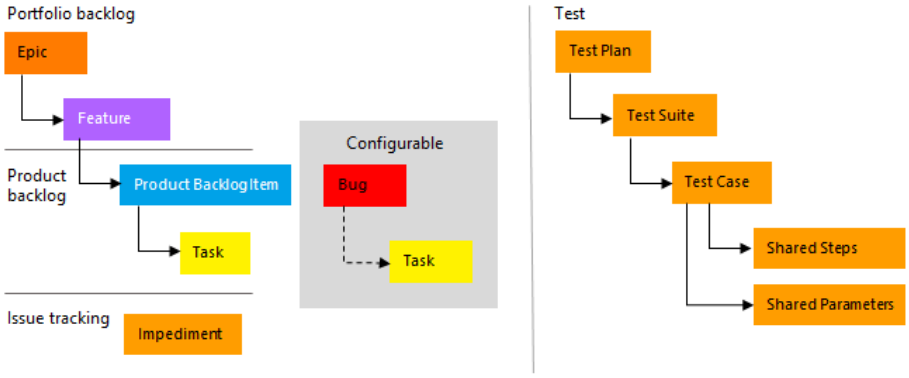


Figure 2.2: Visual Studio Team Services work item hierarchy [7]

Most ESDV teams use Team Foundation Version Control [2] to manage their code; others use Git, but this is only a fraction of the teams.

Work Item Tracking

Work item tracking allows users to set and monitor requirements, tests, bugs, tasks and features. Work items are meant to assist developers and managers by providing transparent communication between development teams and stakeholders in terms of planning and progress. A backlog, or Kanban, board is provided to users to manage their work items. The hierarchy of work items is shown in in Figure 2.2.

A standard ESDV convention is to assign/create a minimum of four Product Backlog Items (PBIs) for each sprint. However, the preference is that more PBIs are created. The size of a PBI is expressed in unit less-numbers called story points. Teams assign sizes to PBIs collectively, with an estimation technique like planning poker [42]. During planning poker, each team member estimates the effort by picking a face down card. After everyone has picked, team members discuss their picks and decide on an estimation. Bugs can also be part of the sprint and should be treated as PBIs, i.e., bugs are assigned a set of tasks and use the same planning technique.

Story points assigned to PBIs and bugs follow a Fibonacci sequence. Product backlog items, bugs and features are categorized into one of five states: new, approved, committed, resolved and done. Tasks can be categorized into one of three states: to do, in progress and done.

Microsoft Visual Studio Team Service Tasks

In a normal scenario, every PBI or bug contains one or more tasks, as visualized in Figure 2.2. The number of tasks assigned to a team depends on how a team interprets the definition of a task, PBI or bug. The task, like other work items, has a certain number of input fields.

[2] https://www.visualstudio.com/en-us/docs/tfvc/overview#team-foundation-version-control

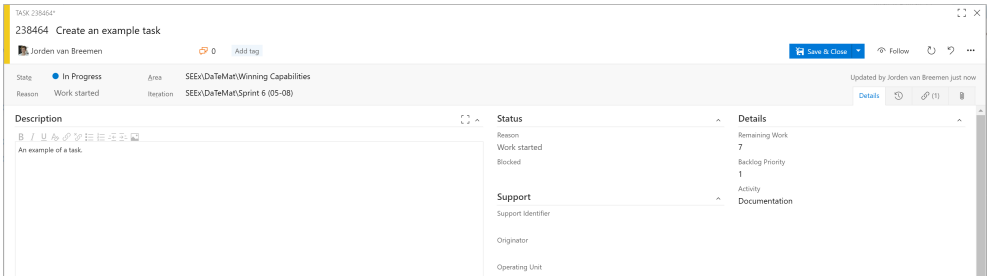


Figure 2.3: A task in VSTS

Figure 2.3 shows an example of such a task. A task belongs to a bug or PBI and should only have a single purpose. A task should not last longer than 15 hours; a common duration of an ESDV task is approximately four hours. Each task should emphasize a single explicit activity, such as writing documentation, learning or development.

Every task can have changesets that provide information about modified files and the exact content changed in every file. To create a connection between a changeset and a task requires certain tooling (such as Visual Studio or plugins). This connection is only consistently used in a subset of ESDV projects, and its consistent usage is one of the primary selection criteria for a project; without it, there is no relationship between the modified code and task. Thus, a *development task* has at least one changeset assigned to it. Without any assigned changesets the task could have a different objective that does not involve changes to code.

The remaining work fields are used to describe the time it will take to complete the initial or remaining effort, in hours, to complete the task. Development teams use this field for most their tasks. Our empirical definition of a task effort estimation originates from the available entire in this field and those of the story points assigned to a PBI. There are several ways to define an estimation using this field; in this study, we define an empirical estimate by the value that the user assigned before the task was put to in progress on the agile board. Revising earlier estimates before starting the task or changing the field during the task are common practices. When this occurs, the estimated time increases; therefore, we also consider a maximum estimated effort variable, which is the longest period an effort assigned to a task can take.

2.3 Project Selection

The data integrity of the projects within ESDV differs greatly. To consider a project eligible for analysis, it needs to meet certain criteria:

Criterion 1: Traceability To find files that were changed in a task, users need to assign a changeset to a task. However, this connection is not mandatory. Projects are only considered when they connect or commit to at least half of their development tasks. Half of the tasks is a relatively high number of reports, because only 15% of the projects meet this

threshold.

Criterion 2: Effort Estimation Consistency is required when performing an effort estimation. Every selected project needs to have a realistic effort estimate for at least half the tasks. The team is expected to perform sprint planning and assign a fitting number when doing task estimates. We ensure that this criterion is met by manually inspecting the recent estimates of each project. When recent tasks show promise, we look for the time when the team started consistently assigning effort. Data are only considered beyond that point.

Criterion 3: Development process Not all projects have fully adopted the DevOps or Agile ways of working. To distinguish this study from past work, it is essential that the teams reached a certain level ASD maturity. To validate this level of maturity, we choose projects in collaboration with the software engineering excellence team of ESDV because, this team is responsible for the deployment and maturity of DevOps.

Criterion 4: Duration and Size Some of the projects are short-lived or have just started development. A project must have at least 200 completed development tasks with a relationship to changesets before it is considered of significant size. After manual inspection, we noticed that after 200 development tasks, projects reached sufficient size and maturity. In these cases, they were at least six months old and produced a significant amount of code. Furthermore, this provided a dataset with a sufficient size to make it eligible for analysis while still getting significant results.

Four projects meet the criteria. Most projects were dismissed because they either lacked effort estimates or were inconsistent in reporting the relationship between commits and tasks. None of the projects that meet the criteria has high turnover, which means that most of the projects have consistent development teams. Some projects meet the other criteria but have just started development. Hence, their size ensured that the resulting data-set would be insignificant.

One of the selected projects is very large; thus we consider three separate development teams for this project. A complete overview of selected projects and teams are shown in Table 2.1. The project names are omitted for privacy reasons.

Table 2.1: Project Statistics

	Completed Tasks	Fixed Bugs	Status	Duration in years
Project 1	8292	4268	Maintenance	5
Project 2	3029	287	Development	2+
Project 3	1304	116	Development	1-
Project 4 Team 1	5851	123	Maintenance	2+
Project 4 Team 2	8514	147	Disbanded	3
Project 4 Team 3	2038	16	Development	1

Project 1 was deployed three years ago; however, it requires constant updates to meet

the constantly changing demand. Hence, development has been ongoing for five years. Because the project has been deployed for such a long time, the number of bug fixes is higher than those of other projects. On average, the team consisted of ten developers, but during the last year, it has been shrinking to about five.

Project 2 and Project 3 are still in development by teams of of six to ten developers. Project 4 is also deployed and in use by many users. All the selected teams are primarily involved in maintaining the state of the project and fixing bugs.

The number of bug reports between Project 1 and Project 4 is very different, even though both projects have been deployed for a long time. This has to do with how a project defines a bug. Project 1 defines a bug as a small problem, best comparable to a task, while project four defines it per the product backlog item level.

The developers of the teams are often co-located, while project management and architects are frequently located at a separate (geographical) location. The programming languages of the projects is primarily C#. Some projects use additional languages for web development, primarily JavaScript and TypeScript.

2.4 Data Filters

Regression is sensitive to extreme values [24]; thus, it is important to remove unrealistic outliers. Teams create tasks for anything that requires effort, such as learning new skills, documenting, investigating, developing and reviewing. Therefore, many tasks are not considered development tasks. We consider a task a development task if it has a changeset associated with it that modifies source code. Non-development tasks or PBIs are not considered.

We consider the state of changes *before* any modification is made; however, developers are also going to add new files. We require at least 80% of the modifications are made to already existing files and that only twenty percent can be made to new files. In this scenario, we define 'modifications' by the number of lines added or deleted to the files. This should include when the executed task maintained the existing code. 80% is chosen because this large majority of the modification will portray a scenario in which maintenance work was likely the primary incentive of the task.

Not every user performs realistic effort estimates for each task; hence, many tasks are available with no effort or high amounts of effort assigned to them. There are also tasks that do not have any effort assigned to them. Removing these cases and non-development tasks has a major impact on the available tasks (see Table 2.2). The remaining data for some projects, primarily for project two, are only a fraction of the original. But out of all the projects available, these were still the projects that are most worthy of consideration.

Additional filtering includes tasks that only have a minor involvement in development. In some scenarios, many of files included in a change are not related to the development process. When the proportion is more than ten percent, these tasks are inspected manually. When the great majority of the task revolve around development, or the extra files consist of primarily meta-data (e.g., project files), the tasks are added to the set.

Table 2.2: Tasks after filtering

	Total Tasks	Development Tasks	Tasks After Filters
Project 1	8292	2683	546
Project 2	3029	998	158
Project 3	1304	471	303
Project 4 Team 1	5851	1363	723
Project 4 Team 2	8514	2929	930
Project 4 Team 3	2038	732	360

2.5 Metrics

Studies indicate that (code) metrics are related to the maintainability [17, 65, 53] of the code. We hypothesize that developers benefit from analyzing quality metrics prior to their estimation so they can incorporate this knowledge in their estimation, increasing the estimation accuracy. If the developer does not consider code quality, the estimate may be inaccurate because the realization of the task/PBI requires more time than estimated.

Quality via statistical analysis tools can also be consistently defined for every task and do not rely on bug reports from users or developers.

2.5.1 Technical Code Quality Metrics

We use *Understand*[3], a tool that calculates static code quality metrics, to obtain a wide variety of code quality metrics [69]. The version we used (4.0) can calculate 107 metrics. A complete list can be found in Appendix A. The list includes a wide variety of commonly used metrics to define maintainability, such as complexity, coupling, volume and unit size metrics [17].

Most of the metrics are calculated for files, and some of them are specified for classes, namespaces and methods. To reduce these methods to a file level, we consider the mean and maximum of the method metrics in a single file. Even when the file is left unmodified, it can have an impact on whether the code is understood. The mean gives an indication of the general overview of the quality in that file. For a small number of metrics we also consider the minimum, because the maximum does not have to indicate the poorest quality.

In all the files considered, more than 90% of the code that contains a class is written in C#. Because the majority of these files only consist of a single class, only the minimum and maximums values of a class are considered.

Whenever a development task is encountered, the metrics of all the changed files are computed before actual changes are made to them. That is, the state of the project is determined before the changesets in the tasks are applied. The output of *Understand* is then stored for every file that was modified during the task.

Because an output from one tool can be unreliable, random results are extracted from the database and validated using Visual Studio 2017. While Visual Studio is incapable

[3] `https://scitools.com/`

of calculating all the metrics produced by *Understand*, the most fundamental metrics are present. Using R, we compared the calculations of the metrics from both tools and found no difference between the results from *Understand* and Visual Studio 2017. What Visual Studio 2017 can do that, *Understand* cannot is calculate via the test coverage. Hence, we can use Visual Studio output to calculate and export this metric.

2.5.2 Process Metrics

Rahman and Devanbu [69] conclude that process metrics are a better way to learn a defect prediction model than code quality metrics. Because defects are frequently associated with quality (Section 5), we choose to include process metrics. Also, during the interviews, we find that developers find experience an important driver during their estimations. Research also shows that experienced developers are better at maintaining the code [51]. Some of the process metrics can express the experience and familiarity a developer has with a certain file. Figure 2.3 shows an overview of a selection of these metrics.

Table 2.3: Process metrics [69]

Name	Description
COMM	Commit Count
ADEV	Active Developer Count
ADD	Total Lines Added
DEL	Total Lines Deleted
OWN	Owner Lines Added
MAJ	Major Contributor Count
MINOR	Minor Contributor Count
SCTR	Code Scattering
OEXP	Owner's experience
EXP	Contributor experience

COMM identifies the total number of commits/changes to the file. ADEV is the unique number of developers that modified the file. ADD and DELL are the total lines added and removed to the file up until that point in time. OWN identifies the total lines added by the owner, the person with the highest number of contributions to the file. MAJ is the number of contributors who contributed more than five percent to that file. This is considered for both lines and total number of commits. The minor counts the number of developers who contributed less than five percent. SCTR is the normalized deviation of the changes from the center of the file. OEXP measures the experience of the file owner via total lines contributed to the whole project during the change. EXP is the same but for the developer who introduced the change to the file.

We further extend the process metrics to metrics related to the size of the change. These metrics are a subset of the distribution shown in Moser et al. [62]. Commonly, in agile estimation procedures, size is the primary value that influences the estimation [31]. Although developers might not be able to express the size of their changes via an exact metric (like

LOC), we hypothesize that they do consider the size of the change during estimates. Table 2.4 lists these metrics.

Table 2.4: Change metrics [62]

Name	Description
Bugfix	File involvement in bugfix
LOC_ADD	Lines added to a file in commit
MAX_ADD	Maximum lines added in all commits
SUM_ADD	The sum of all lines of code added in a task
AVG_ADD	Average lines added per commit
LOC_DEL	Lines deleted to a file in commit
MAX_DEL	Maximum lines deleted in all commits
AVG_DEL	Average lines deleted per commit
SUM_DEL	The sum of all the lines of code deleted in a task
CHURN	Sum of lines added - sum of lines deleted for all commits
MAX_CHURN	Maximum churn for all commits
AVG_CHURN	Average churn for all commits
MAX_CHANGE	Maximum number of files involved in one commit for a task
AVE_CHANGE	Average number of files involved in one commit for a task
AGE	Age of a file in weeks
WEIGHTED_AGE	See formula 2.1

Bugfix represents the number of times a file was related to a bugfix up to that specific point. This also includes tasks that are involved in a bug, which conform with the structure shown in Figure 2.2. LOC_ADD is the lines of code added to the file during the commit in the task. MAX_ADD includes the maximum number of lines added in a single commit during the lifetime of the file, while AVG_ADD is the average number of lines added over all time. SUM_ADD is the total number of lines added to a task. CHURN is simply the total lines added minus the lines removed over the lifetime of the file. Thus, the weighted age is calculated by equation 2.1 [62].

$$Weighted\,Age = \frac{\sum_{i=1}^{N} Age(i) * LOC\_ADD(i)}{\sum_{i=1}^{N} LOC\_ADD(i)} \tag{2.1}$$

In addition to these metrics, we also determine the mean, max and sum of all the files that were changed during a task. Other than default files we also consider the mean, max and sum of source files (e.g., excluding doc files). Finally, we consider the total number of changesets that were applied to finish a task.

2.6 Reducing Metrics to a Task

There is only a single effort estimation for a task. To have comparable impact from all tasks, we should represent all the metrics on a task level, and not individual for every file.

Table 2.5 shows the average number of files changed in every considered task. File changes contains the total number of files that were changed across all the tasks. Because an effort estimation is done at the task level, we only have a single effort estimation for a task. Therefore, the data should be further reduced to serve as input for the regression model. The reduction consists of calculating the mean and the max [69] for every metric of each file changed in a task. There are metrics that are better represented when summed up, for example, when considering the lines added/removed in every file change in a task.

Table 2.5: Considered files for every project

	Considered Tasks	**File Changes**	**Average files/task**
Project 1	546	6682	12.2
Project 2	158	1090	6.9
Project 3	303	2474	8.2
Project 4 Team 1	723	3726	5.2
Project 4 Team 2	930	14031	15.1
Project 4 Team 3	360	3660	10.2

After these steps, the input for the regression model consists of one row for every task with an estimated effort and the means and maxes of code quality and process metrics (Figure 2.4). Process metrics consist of the means, max or sum of the metrics. The number of rows will now equal the number of considered tasks. In Section 2.10, we discuss how to further process these variables and how they might serve as an input for linear regression. We also evaluate two separate inputs for the regression model, one containing only the means and another containing only maxes, to verify that these models do not provide stronger results (due to lower multicollinearity). We report these results when they are more significant than the combination of both the means and maxes.

Please note that bug tasks often involve many other factors than those described. Therefore, we do not consider bug tasks in this context.

In an attempt to obtain stronger results, we also considered other methods. One of the more notable is a binary approach, in which a file is considered a bad quality when a certain metric threshold is met. If no scientific or industrial [1, 15] definition of such a threshold can be found, we define it based on own experience and by inspecting the available data. For example, if the number of fields declared in a class exceeds a threshold of twenty, then it is considered bad quality, and the metric of the file level is flagged as 1. We also attempt a methodology in which a certain number of metric thresholds must be exceeded, e.g., twenty percent of metrics must be of bad quality before the file is flagged as having poor quality. Because the results were either comparable or weaker, no further attention was given to these representations.

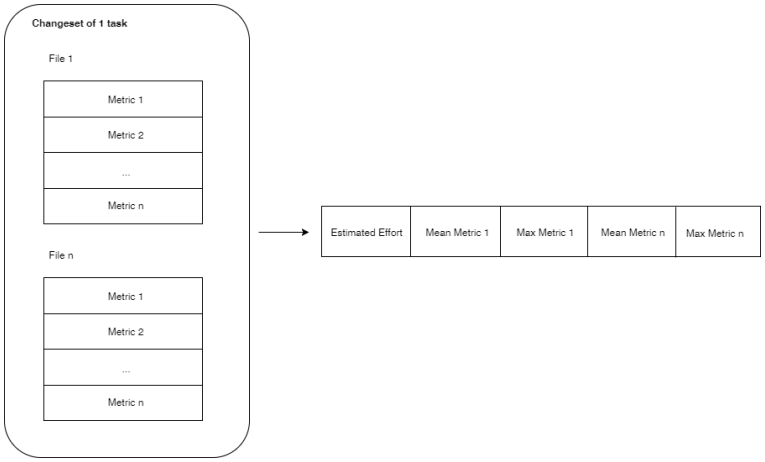


Figure 2.4: Representing one task and its changes on a single row

2.7 Reducing Metrics to a Product Backlog Item

A task does not incorporate the confounding work required to realize development, like an investigation or a learning task. A PBI, however, will contain all the tasks required to realize the implementation. To include these dependable tasks, and to assess if they are impacted differently than code quality, we also consider the effort estimations of a PBI.

The quality of the code will be the average, max and sometimes the sum or minimum of the metrics in a PBI. Hence, the technique is the same as for a task. The total effort will be the sum of the estimated efforts of all tasks that belong to the PBI. Often separate tasks are created for investigation, documentation and research required to complete the PBI. These tasks belong to the same PBI but will not be incorporated in the task method. This technique includes these additional hours needed to complete the work. The files that belong to the PBI are all the files committed to the tasks of the PBI and all the files committed to the PBI (it happens from time to time that one can also assign a changeset to a PBI). We also consider a scenario in which we compare not the sum of the tasks, but the story points as effort. Table 2.6 shows the total available data for this method.

Table 2.6: Total PBIs and considered PBIs

	Total PBI	Considered PBI
Project 1	1609	223
Project 2	230	61
Project 3	143	77
Project 4 Team 1	108	61
Project 4 Team 2	204	72
Project 4 Team 3	31	16

The reason for the reduction in available data is comparable to that of the tasks. Not all PBIs will be related to development and, in some cases, not all teams will be consistent

in assigning changesets to PBIs and their underlying tasks. The method to calculate the actual effort (Section 2.9) is not applicable here, primarily because one of the criteria for both approaches is that no tasks can be executed at once. Thus, if we were to consider these tasks, they would be near impossible for PBIs because it is very common to work on multiple PBIs (and hence multiple tasks) simultaneously, leaving little, if any, data to work with. We also manually inspect PBIs that appear to have a very small number of file commits in accordance with the total effort, thereby removing any unrealistic scenarios from the dataset.

2.8 Empirical Data Collection: Tracer

This section briefly describes an analytic platform developed for Shell that is used to extract, store and analyze data to answer research questions. The platform helps visualize VSTS meta data and assists the software engineering excellence team with supplemental analysis regarding VSTS use. In this chapter, we discuss the need for this platform, how it is designed and how the data are collected. This analytic platform is called 'Tracer'.

2.8.1 The Need for Data Collection and Visualization

VSTS has a reporting capability, but it is still under development and the scope is still quite narrow. ESDV requires a visual method of monitoring the behavior in VSTS, a process that is currently done manually. This includes general overviews of sprints, backlogs, deployments, defects, tests and other metrics, but also errors and noncompliance, such as the previously addressed missing effort fields. Teams that only make limited use of the features available in VSTS frequently do this unintentionally. ESDV can then identify these teams to provide them with appropriate coaching and training. Please note that these metrics and charts are not related to software quality, but focus more on the software development process. An explanation and visualization of the charts can be found in Appendix B.

Other than a visual need by ESDV, there is also demand from a research perspective. When one considers a large number of work-items, Microsoft's REST API does not immediately support the retrieval of the effort fields. The platform must re-structure the data so it can be easily and quickly accessible. For example, process metrics require information about every individual file version. This would take an extraordinary amount of request in the default REST API. Furthermore, we require a method that allows statistical analysis tooling to calculate metrics for every version of the project, and store these states in a database.

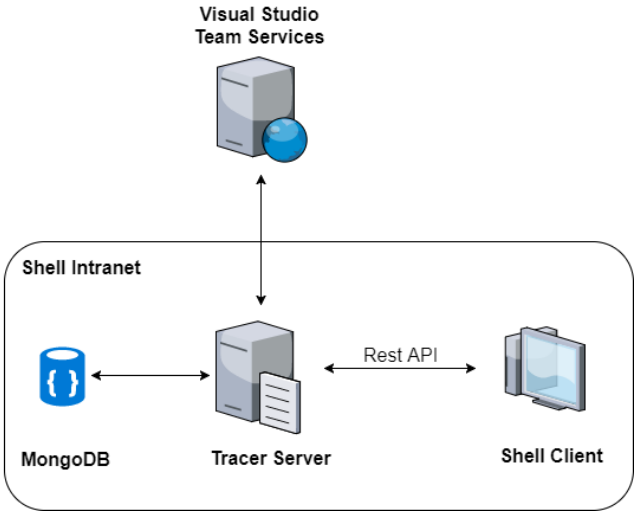
2.8.2 Tracer and Metric Calculation

Visual Studio Team Services features a REST API[4] that enables anyone with access to the project to extract nearly all available information in some form. The original platform architecture was primarily a front-end application that visualized the data available through

[4] https://www.visualstudio.com/en-us/docs/integrate/api/overview

this API; however, when implementing more complex metrics that require insight into revision histories of the work items, the data structure and performance of the rest API were not sufficient or suitable. Therefore, the decision was made to incorporate a back-end into the architecture to restructure and store all relevant data from VSTS. This architecture is visualized in Figure 2.5.

Figure 2.5: Simplified Architecture of Tracer



For more (implementation) details about the platform, and some of the charts it can visualize, please see Appendix B. Tracer is capable of storing all VSTS data we need, meaning the complete estimation history but it is also capable of calculating all the metrics previously described.

There are two methods to calculate the metrics. One simply pulls every version of an individual file to the server and calculates the required metrics from the tooling. However, metrics like test coverage require the current state of the project. Therefore, a second approach simply goes through the changeset history of the project and calculates the metrics of the modified files for every individual changeset.

2.9 Actual Development Effort

To answer how accurate the estimations are, we need to devise a method to calculate the actual time spent (actual effort) on the task. This actual effort is compared to the estimated effort that measures the estimation accuracy. Using multiple linear regression, we determine if code quality was poor for the prolonged development times. This will inform us about the importance of code quality considerations during agile effort estimations.

Certain criteria should be met before a task can be eligible for the calculation of the actual effort. Some developers can work on multiple tasks at once. In such scenarios, it is inaccurate to consider the parallel tasks because one cannot accurately determine how many

hours were spent on the individual tasks. Therefore, all tasks in the in progress state of the Kanban board simultaneously are excluded from the considered tasks.

There are some scenarios in which a developer could drag a task to the 'in progress' state and quickly thereafter complete another task that was already in progress. When such scenarios occur and the overlap only lasts for five minutes, the tasks are still considered. We consider five minutes a sufficient amount of time to maintain tasks but not to perform any actual development. When a task is very short-lived (under 10 minutes) we also exclude it. These tasks have a very high likelihood that actual development work was done, but that the work was completed before putting it in the in progress state. Something that is also very probable for tasks that have commits assigned to them before they are set to 'in progress'. Hence, such tasks are also excluded.

When tasks are not completed on the same working day, but on a following day, many uncertainties arise. For example, we cannot know exactly when the developer stopped working or when he or she started on the next day. To tackle this problem, we considered two approaches that both make use of user activity. We calculate user activity by observing any measurable VSTS activity with a timestamp. The primary source of this data is the platform developed in Section 2.8. Examples include interaction with any of the work items (e.g., changing the state, estimate or comments) and committing files. We then combine the activity of a single user, over several sprints, to determine the length of time a user is usually active. Figure 3.3 shows an example of the activity of two individuals.

Activity is computed for every individual user. To determine the activity of the user, we consider the highest and lowest ten percent of activity sets. Next, the median time of these sets is calculated to determine the range of activity of a user. We choose the median, not the mean, because it is less influenced by large outliers in the data set. Outliers consist of people who perform actions beyond their normal working hours. This method was validated by manually inspecting the activity of seven users, for which the actual times of activity were known. These active hours are also confirmed for four of the interviewed developers. Whenever the office hours deviate greatly from the actual hours, the methodology is revised. We use these boundaries in two separate methods that we assess in Chapter 3.

We consider two methods to calculate the actual effort spent on a task. In the first method, we simply exclude any tasks that was not completed on the same day. Any task completed at an irregular time is also excluded. To determine what tasks should be part of the subset, we use the previously determined periods of user activity. We apply a one-hour grace period; tasks started until an hour before the start time, or finished up to an hour after stop time are included. We subtract and add an hour because the working hours are not completely consistent. One hour is just enough to exclude any tasks a developer might complete at irregular hours. Figure 2.7 shows an example of two users who are active from about nine to five and four to three.

This procedure results in a significant reduction of available tasks. In an attempt to increase the subset, we try a second approach. Here we also consider tasks not completed in a single day. To do so, we simply use the active hours to determine what time was spent on the task. Figure 2.8 shows an example.

Because this approach likely has a reduced level of accuracy, primarily since users are not always strictly active between the calculated time boundaries, we consider it separately

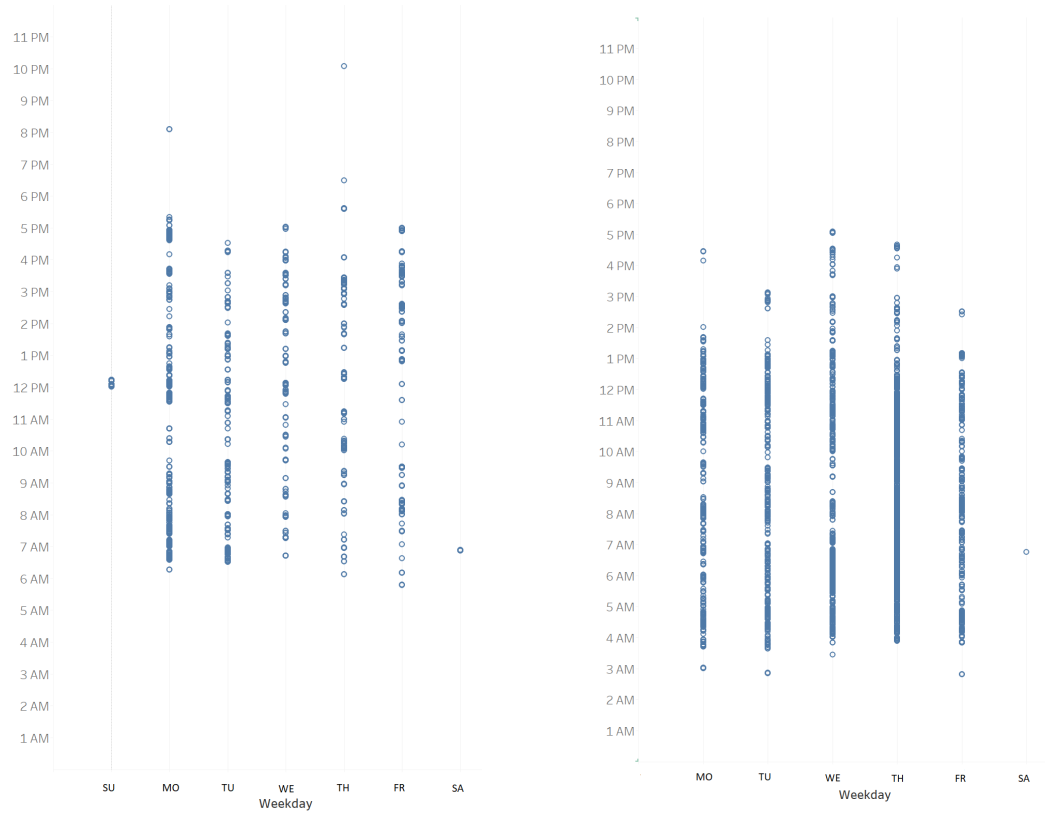
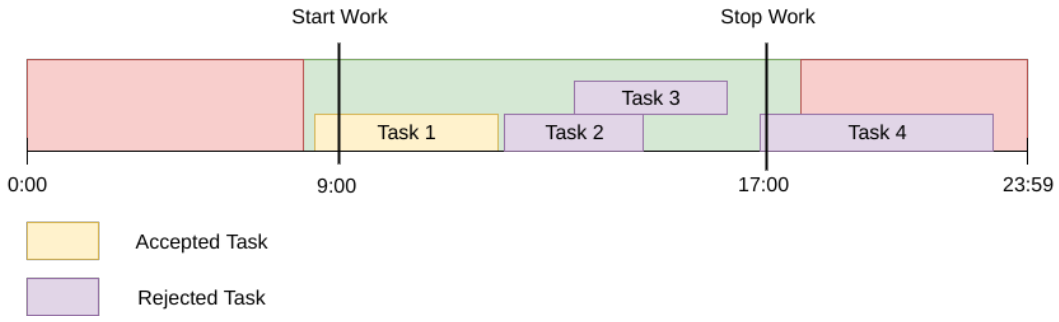


Figure 2.6: Activity for two users

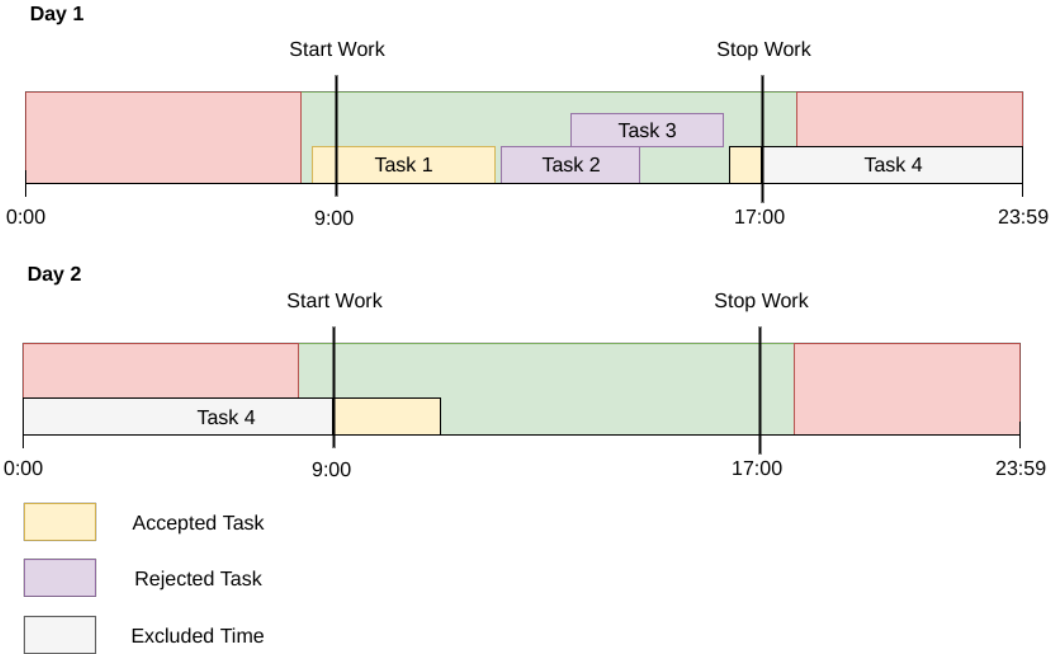
Figure 2.7: Simple one-day task selection



from the single day method. In an attempt to include certain scenarios of inactivity, such as lunch breaks, we also consider two methods. One tries to derive the lunch break times from user activity. For example, there are small gaps in some of the days in Figure 2.9. The other method relies on feedback from users about when they usually take their lunch break in that specific office, and on asking them how long they take on average.

To determine if a developer was active on a certain day we look at his or her activity on that specific day. Before we consider a developer active, the developer should at least show

Figure 2.8: Next day task selection



some form of activity on that day's activity log. If the user was active that day, we include it in the calculations; if not, we exclude that day from the calculations, in the sense that the hours of that day are not included in the calculation of the actual uptime. When tasks were modified during the weekend, we choose to manually inspect them, as weekends are days off for most Shell staff.

Project 1 also has a custom completed work field attached to every task. This field represents the actual work done, in hours and is filled in by the person who executed the task after completion. Because this field has only been in use very recently (in the last three months), not enough data are available to use it as an evaluation of our methods, primarily because the methods do not seem to work very favorably for project one.

To see how the actual hours compare to the estimated effort we calculate the correlation between the actual effort and the estimated effort. We do not consider this to be a method of evaluation because deviations from the estimation are expected. We do however want to discover how they compare to see how accurate the estimates are.

2.10 Multiple-Linear Regression

Table 2.7 shows the many metrics proposed for regression. Many predictors are very similar in nature (Appendix A). These comparable predictors correlate strongly and, if not handled, will greatly influence the model. We consider two separate approaches to reduce multicollinearity [56]. The first approach, principal component regression, does not require knowledge of the predictors. This method is applicable when we do not know what metrics

the developers prioritize. The second approach requires knowledge and makes assumptions about the predictors, which will be more applicable after we interview developers and ask them about what metrics they consider important.

We consider three separate models for regression, one for every set of code metrics described in Section 2.5. The third combines all these metrics into one as a complete definition of code quality.

Other than regression we also simply evaluate how the predictors correlate with the dependent variable. We consider $\rho = 0.3$ a weak correlation, $\rho = 0.5$ a moderate correlation and $\rho = 0.7$ a strong correlation [73]. Linearity between the relationships is not necessary, so the Spearman correlation test is applied.

Table 2.7: Total metrics that serve as input

Metric Model	Unique Metrics	Total Metrics
Code Quality	108	204
Process Metrics	26	42
Combined	134	246

2.10.1 Principal Component Regression

Principal component regression (PCR) is based on principal component analysis. This technique overcomes the multicollinearity problem [56], another problem of similar predictors, by excluding low variance principal components as the input for regression. PCR also visualizes the regression results and thus enables reduction of the dimensions of the model. This means we can select the number of predictors required to obtain the most significant results.

The `pls` [5] package in R allows an easy and complete implementation of PCR [6]. Because the PCR method from `pls` does not scale, we initially normalize the metrics and effort. In the first step, we apply PCR to all the predictors and evaluate the outcome. When plotting the R^2 values, we evaluate how many predictors are required to achieve a consistent result. We then reduce the total considered predictors to the influential predictors and assess the impact of every individual predictor using the Jackknife test [36], a re-sampling technique. As an example, consider figure 2.9, in which the number of predictors seem to increase the accuracy.

This increasing effect, with an increasing number of components, is prone to overfitting. The model keeps benefiting from the additional variables because they are a perfect fit for the given input. To avoid overfitting effect, we apply cross-validation, which takes chunks of the data set and validates the integrity of the model on those sections. Figure 2.9 shows the effect of cross-validation on the same data-set as Figure 2.10.

This indicates that while the results in Figure 2.9 were not that favorable to begin with, there seems to be a very limited use for predictors. Also, the model appears to suffer greatly

[5] `https://cran.r-project.org/web/packages/pls/index.html`

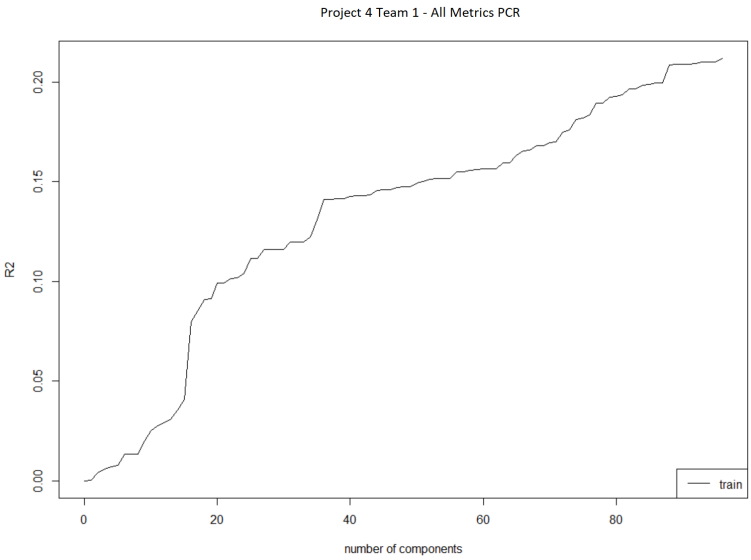


Figure 2.9: R^2 PCR for Project 4 Team 1 Combined Metrics for RQ1 - Task

from over-fitting, as indicated by the negative R^2 values. As expected this effect increases as the number of predictors increase. The R^2 value we report for PCR regression are the highest obtained values by cross validation.

In this chart one can see spikes (or points with a high gradient, like metric 18 in Figure 2.9) if there are predictors that greatly influence the model. We can determine the number of influential variables using these spikes.

2.10.2 Predictor Selection

McIntosh et al. [60] analyze the impact of modern code review on software quality. While their approach and definition of quality differs from this study, they provided a methodology, that includes linear regression that is applicable to our setting. In this section, we summarize and modify (when necessary) their approach to fit the setting of this study.

To reduce multicollinearity, we first identify which predictors correlate strongly with each other, according to the Spearman rank correlation test (ρ). We choose Spearman because it is resilient to data that are not normally distributed [60]. To visually identify correlation clusters, we use the `varclus` function from the `rms` package. We draw a dotted line to identify any clusters that exceed a correlation value of $\rho = 0.7$ [19]. Figure 2.11 shows a example for the process metrics of project 1 with the estimated effort.

Two choices are applicable when choosing the most fitting variable from the clusters. If we possess knowledge to manually identify the most influential metric, based on the results from the interviews or domain knowledge of the project or metric, we choose it manually. If we are unable to choose one manually, we pick the variable that correlates most strongly with the dependent variable.

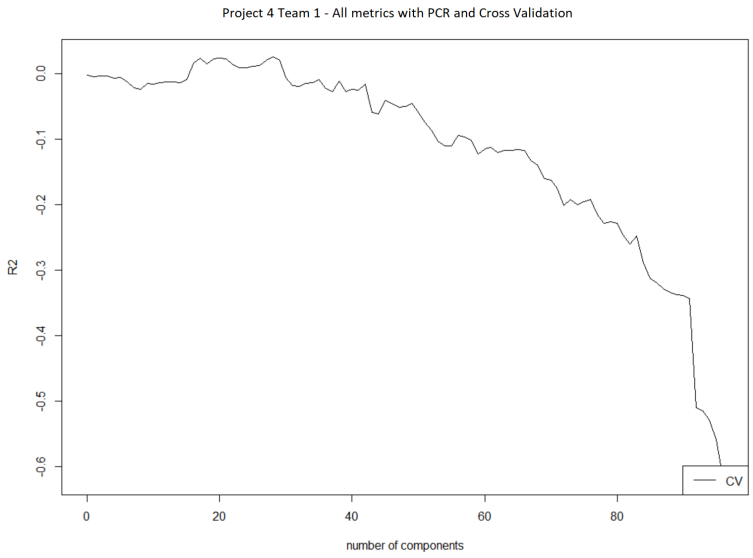


Figure 2.10: R^2 PCR with Cross Validation for Project 4 Team 1 Combined Metrics for RQ1 - Task

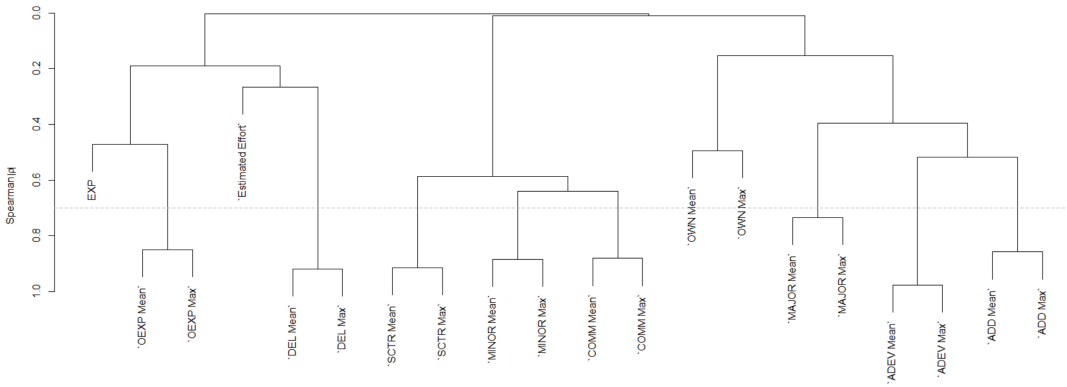


Figure 2.11: Hierarchical Spearman clustering for metrics with estimated effort

The outcome of Figure 2.11, after the selection procedure can be seen in Figure 2.12. Table 2.8 presents a complete overview of the remaining predictors for every model. The great reduction in predictors (especially for code quality) indicate that there was a high correlation among variables. This also tells us something about the high amount of redundancy among the technical code quality metrics. While *Understand* can calculate many code quality metrics, most of the metrics seem to correlate strongly. One primary reasons is that the tool tends to calculate many variations of the same metrics, such as variations on cyclomatic complexity, number of methods, number of comments and lines of code, that produce results that are strongly correlated.

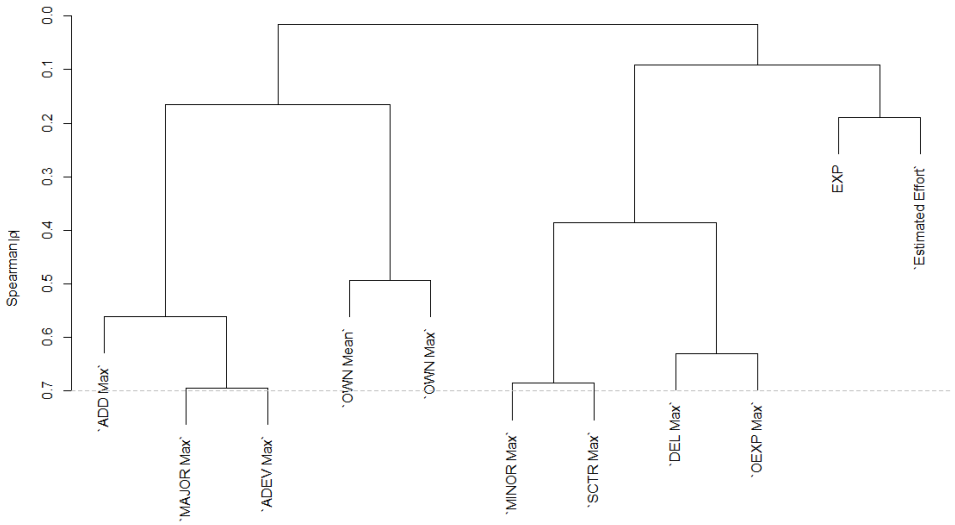


Figure 2.12: Variables from Figure 2.11 after the selection procedure

Figure 2.11 shows that the mean and the max of all metrics, calculated according to Section 2.6, frequently exceed the correlation threshold of $\rho = 0.7$. It is difficult to determine which would have a more significant effect on the dependent variable. The general approach is to pick the variable that correlates most strongly with the dependent variable. To verify that this selection procedure does not have a negative effect we compare it to the output that the model would produce if only means and maxes were considered as input. In all cases these results are either equal or less significant, we do not report further on these results.

Table 2.8: Considered metrics after selection and tasks for every project

	Tasks	**Code Quality**	**Process**	**Combined**
Project 1	566	6	13	19
Project 2	161	9	12	16
Project 3	307	7	12	16
Project 4 Team 1	528	6	12	13
Project 4 Team 2	932	7	11	16
Project 4 Team 3	116	5	11	15

To further reduce the effect of overfitting the regression model, we define a limit for the degrees of freedom. We initially define a total available budget for the degrees of freedom.

Harrel et al. [60] suggest a budget of $\frac{n}{15}$ where n is the number of considered tasks.

After removing the correlated and redundant variables, we to identify the most relevant variables to stay within our allocated budget. To identify relationships between these selected variables we use the *spearman2* function from the *rms* package. This function can calculate the ρ^2 value that is then plotted to identify strong non-linear relationships between variables. The earlier calculated budget is then distributed among the relevant variables, in which larger ρ^2 values are allocated more degrees of freedom.

We use two regression functions, the default *lm* regression available in *R* and *ols* from the *rms* [6] package, because each has some advantages we can use. Both assess the fit with the default R^2 and the adjusted R^2. Adjusted R^2 reduces model bias by penalizing models based on their degrees of freedom spend. Because we reduce correlation and allocate the degrees of freedom, the R^2 and Adjusted R^2 are not too distant from one another. To assess the significance of the R^2 we also report the p-values for every fit.

As a final measure to prevent overfitting, we apply a method that compares with cross validation. We subtract the bootstrap calculated optimism [37], which is implemented using the following five steps [60].

1. Select a subset of bootstrap samples from the original data set;
2. Fit the model using this subset and the same variables / degrees of freedom as the original input;
3. Calculate the adjusted R^2 values from the original model;
4. Assess the difference between the adjusted R^2 and the original model;
5. Repeat these steps 1000 times using different bootstrap samples;

By comparing the output of the steps with the original input we can assess the overall stability of the model. If they only differ slightly, we consider the model to be a stable fit. These steps are made using the *validate* function from the *rms* package.

To evaluate the impact of every explanatory variable we use two techniques. One using the χ^2 maximum likelihood test [55] using the 'drop one' approach [40, 60]. Here the model is initially assessed using all explanatory variables and then again without a certain variable. The difference in performance then identifies the significance of the variable.

Another method to evaluate the impact of predictors applies the *varImp* [80] function from the *caret* [7] package. This function takes the default R regression model as an input and calculates the absolute value of the t-statistic for each model. This is only a second variable assessment, the primary evaluation is done using the χ^2 maximum likelihood test.

[6] https://cran.r-project.org/web/packages/rms/index.html
[7] http://topepo.github.io/caret/index.html

2.11 Interviews

The reasons behind how developers and other experts estimate task efforts, or how they incorporate code quality during their estimations, is hard to deduce only using data analysis on history of source code. Therefore, we choose a mixed method approach, to validate the quantitative results, but also to obtain knowledge about how developers improve code quality, and determine the metrics they consider important. During the interviews, we want to:

- Explore how developers define and maintain code quality;
- Explore how code quality and code quality metrics influence effort estimates;
- Explore how developers evaluate their code quality and if they consider these evaluations while estimating;
- Identify factors that influence the estimates of a development task;
- check if developers perceive to frequently encounter scenarios where they require more effort than estimated due to poor code quality;
- Ask what metrics they consider important;
- Explore the impact of (poor) code quality on actual effort.

The candidates consist of two Scrum masters, five senior developers, a quality expert and the ESDV quality lead. The interview with the quality expert serves a different purpose than the other interviewees; to acquire adequate suggestions and draw accurate conclusions, because we need to validate that the projects follow the general guideline set by ESDV. Table 2.9 shows an overview with all the interviewed developers and some background information. We refer to the participants in this section using these abbreviations.

Table 2.9: Interview Participants

Identifier	Project	Role	Experience in years
P1	Project 1	Senior Developer	4+
P2	Project 2	Senior Developer	5+
P3	Project 3	Quality specialist/Dev	7+
P4	Project 4 Team 1	Senior Developer	5+
P5	Project 4 Team 2	Scrum Master	5+
P6	Project 4 Team 2	Developer/Analyst	3+
P7	Project 4 Team 3	Scrum Master	8+
P8	Project 4 Team 3	Senior Developer	5+
P9	All	Quality Assurance Lead	10+

We plan thirty to forty-five minutes for the interviews, which consist of two parts. Before the begin, we ask for informed consent to record the audio to assure anonymity.

The first part of the interview consists of a semi-structured section [47] that explores all the previously mentioned points. An early version of the interview questions can be found in Appendix C. We encourage developers to speak freely to gain more information and to ask follow up questions [47]. To examine the impact of the metrics, we explain the metrics and ask the developers if and how they impact their estimates or if they use them in another way. We constantly adjust the questions if the answers reach a saturation point.

The second part consists of discussing a set of three development tasks. These tasks are no more than two months old because older tasks seemed to be difficult for developers to recall in detail during the interviews. The first task has an effort estimate higher than six and is of poor quality code. Tasks with poor code quality are identified by lining up all the tasks completed by the developer in the last two months. We then look at the code quality metrics that relate to every task and manually select the task with the poorest quality. The mean of the effort estimates is frequently valued around four, hence we choose six as a threshold to identify a task that took longer than average. However, some projects have a very different distribution. Hence this value (six) might differ slightly for some projects. The goal of this task is to identify if the higher effort estimate had something to do with the poor quality of the task. And, if developers thought the work would require more maintenance effort due to poor code quality, and how developers measured that code quality.

The second task identifies a scenario in which a task of poor quality has a revised estimate. Revised means that the developer initially created an estimate for the task, but changed his or her estimate *before* starting development on the task. The purpose of this task is to derive why the developer revised his or her estimate and if later obtained knowledge about the quality of the code had anything to do with his or her revision. Because these revisions are not a common, some tasks were actually not of poor code quality. Still, we tried to derive the reasoning of the developer as to why they revised his or her estimate. In most cases, developers fill in the effort field after the task has started. For example, if the developer has to halt his or her work for the day, because his or her work day has ended, they can use this field to inform the team about how much of the task they have completed, and how much work remains. On some occasions this value would be higher than the estimate.

For the third task, we search for such scenarios. We hypothesize that someone might have filled in a higher effort in this field because he encountered code with poor quality. From the subset of tasks that meet these criteria we again select the one with the poorest quality.

For every one of the tasks we ask the developer how he arrived to this estimated value, what other factors than quality influenced his or her estimate, and how accurate the estimate was.

2.12 Threats to Validity

The selection criterion in Section 2.3 resulted in only four of many available projects. The reason we chose these projects instead of the others, is primarily because there was a lack of agile adoption or consistent reporting. These criteria resulted in mature and well managed projects that followed the process guidelines correctly. The effect of quality on software development might be stronger in projects that did not achieve this level of adoption. We tried to validate whether the overall quality of the selected projects really is better than other available projects, by considering the quality metrics of all projects in the current state and from several months ago. We were unable to conclude that there is a significant difference between the quality metrics of the selected projects and the excluded projects.

As becomes clearer in Section 2.4, the data and developers seem to lack consistency. Without a certain level of consistency, in both the effort estimations and reporting of the changesets to the tasks, the results will be inaccurate. We mitigate this threat by pre-selecting the most consistent projects available and filtering out outliers (e.g., non-standard times and types of data) (Section 2.4).

One could argue that the effect of code quality is highly dependent on the developer, and that considerations from an agile team is too high level. However, agile effort estimations during techniques such as planning poker are a team effort. All metrics are already developer-specific. To further mitigate this threat we applied the same methodology as described in Section 2.6 for every individual developer. Many of these results did not show statistical significance or did not produce results that would result in different conclusions.

This research uses four large projects within ESDV at Shell Global Solutions and may not be applicable to the whole industry. These projects all follow standards set within Shell and their way of working. The way Agile and DevOps are deployed in ESDV may have a great influence on the study results, limiting the overall scope. However, academics have shown that individual cases can obtain details that research focused on groups cases could not [16, 39].

During this study, we focus on a file. Developers may only consider the methods and surrounding methods they changed. We attempted to identify the exact methods or functions that got changed during a changeset, and then linked these specific method metrics to effort. The automated identification of the changed methods was however difficult to realize. We consider the changes at a method, but determining which methods got changed during a change proved to be very difficult, especially with multiple languages and complicated syntax.

The calculation of the actual development effort imposes several limitations. We do our very best to empirically define them, but this study could never reach the levels of confidence that could be achieved by, for example, a controlled experiment [77]. The method requires assumptions that, while applicable to most data, will not hold for all tasks. We attempted to further enhance the actual effort method by including the Microsoft Exchange used within Shell. This turned out to be impossible due to privacy and authentication policies deployed by Shell. However, we compare the method to a completed work field for one project, and find that the method generates comparable results. Furthermore, we manually inspect the tasks that remain for the actual effort calculation, to make sure that there were

no outside influences.

Understand calculates a wide variety of metrics. However, duplication, design and smell metrics are excluded because the tools that allow these calculations require the .net files generated by C#. Because the projects were very complicated, generating these .net files in the same methodology (for every version of the file) would require very detailed knowledge about how to compile the projects. The tasks would also require significant computing power, knowledge and resources that were not available.

The interviewees only consisted of experienced developers from already mature projects. One could criticize that these developers are all too experienced with the source code they modify. To mitigate this threat, we attempt to vary the experience levels of the developers.

The interviews focus on the effect of code quality on agile effort estimations. This can make the interviewee lose sight of what factors, other than those related to code quality, can affect his judgment. We attempted to avoid these scenarios by first asking open and project-specific questions about effort estimations without mentioning code quality. We further invited the interviewees by email without mentioning such keywords.

Chapter 3

Results

3.1 RQ1: How do developers consider code quality during agile effort estimations?

We investigate how developers define quality and assess how their definition compares to the traditional definition of quality. Furthermore, we investigate how they attempt to maintain high quality and question if code quality plays a role in agile effort estimation.

3.1.1 How do developers define code quality?

Most participants associate quality aspects to methods that improve code maintainability. The Quality Assurance Lead (P9) defined quality as:

"*I would define source code quality or the internal software quality, by the way the quality is measured, if you are following or breaking any coding standards, are you preforming any static quality analysis, how complex is your code and is the software properly designed?*"

P3 (table 2.9), who is responsible for the quality of the project says:

"*Our primary definition of quality is meeting the acceptance criteria and maintaining an adequate test coverage, we also incorporate code quality metrics using SonarQube at every changeset.*"

The definition of `done` differs in every project, i.e., includes reaching an adequate level of test coverage. SonarQube can detect syntax policy violations, duplicate code and known security vulnerabilities [5]. This tool is popular among teams because it was recently introduced in ESDV by P9. P1 (from Project 1) did not use the tool, nor had he heard of plans of deploying it in his team.

When asked about ways of measuring software quality, all participants from project 2, 3 and 4 mention SonarQube. They (P2-P8) use it by first submitting their code changes and then validating the SonarQube output. If SonarQube warns the developers about issues or shortcomings they fix them after their commit. They do not create a separate task for it. When asked if they incorporate the SonarQube results into their estimates, developers (P2-P5,P7 and P8) often mention they do account for it. P8 states,

"*We take all of the SonarQube results into account. To incorporate the output of SonarQube we add an maximum of one hour to a task estimate. So overall it has a big impact on the effort estimates.*"

When asked what metrics developers emphasize in these (two) tools, P2-P8 mention cyclomatic complexity, test coverage and code smells (SonarQube). McCabe's cyclomatic complexity measure has proven very successful [57], and testing is often seen as an important practice to assure quality in DevOps [72]. Therefore, these metrics are prioritized in ESDV and by the software engineering excellence team.

P4 and P8 also mention they use ReSharper [1] to maintain code quality. ReSharper informs the developer about metric violations, redundancies, run-time errors, compiler errors and code flaws during development. Other participants do not mention any other tools they used to maintain quality. When we asked P9 about what she considered proper metrics to pursue she mentions:

"*Initially if we are breaking any coding violations, like specific language rules and naming rules. We have to validate if we are adhering to these rules. You can also look at the number of issues introduced and code duplication. Because if you just copy paste the code the chances of introducing a bug in a new area is likely also more. Other than that cyclomatic complexity, while not applicable for all code it certainly helps to optimize it.*"

She mentions some factors covered by SonarQube, e.g., complexity and coding, naming and language violations. Duplicate code is mentioned by a subset of developers (P4, P5, P7). P7 mentions that SonarQube is not adequately able to identify duplicate code, and hence they do not incorporate it frequently. Coding violations are covered by SonarQube, which can identify errors such as naming violations. Only one interviewee (P1) mentions code reviewing when asked about code quality:

"*We do not think about the code we are going to change. It does not work in accordance with our way of working. We very frequently review each others work, which helps us retain quality and functionality.*"

This confirms that Project 1 does not use any static analysis tools, but the team members do review, a practice that is associated with positive quality effects [59]. The other teams acknowledge they occasionally review code, but never comparable to the 80-100% review coverage that Project 1 achieves according to P1.

When we ask P9 if she considers metrics a good method to quantify code quality, she answers,

"*In one way yes, because if you have a globally distributed team, you need to collaborative method, like a number, to read quality. To empirically quantify quality really helps the team, because if everyone codes accordingly, standards will not be different from the one person's perspective to another person's perspective.*"

We observed that most of the developers seem to keeps control on the code quality through tools. They optimize for coding violations, complexity measures and test coverage reported by primarily SonarQube.

[1] `https://www.jetbrains.com/resharper/`

3.1.2 How do developers consider code quality during effort estimations?

We asked a series of questions about their task effort estimation approach (Appendix C). We review a specific task with every interviewee that has a high effort but poor quality. We define the poorest quality by looking at all the modified tasks in the last months by the developers, and pick the ones that scored the worst (or highest) according to the metrics. We also considered a task with an estimated effort that was revised to a higher value. We use this task to explore if quality had an impact on their revision.

The planning approach adopted by Shell is planning poker [31]. All developers mention that they perform their planning as a team at the beginning of the sprint. P5 acknowledges that their team often includes the opinion of the technical architect and business analyst during the sprint planning.

P2, P7 and P8 mention that, when estimating task effort, they prefer to pick the highest mentioned estimate from all developers because this is a good method to account for the knowledge gap between team members. According to some developers (P3,P4-P6), the deciding factor in estimates is experience and familiarity with the code. Only P4, P7 and P8 acknowledge they will manually inspect the content of the files before their estimate. All developers acknowledge they think about the files or modules they have to modify. All but P1, P2 and P4-P7 think prior inspection is too detailed, or simply find it tedious. Others think it would add value, but they simply do not do it. As P7 puts it:

"*It is not about thinking about the files, it's about thinking what core things you will change. We don't really go through the lines of the files, rather we look at the functionality and complexity and other things to estimate the task.*"

When asked if static analysis tools were used during their estimation process, none of the developers answered they did so. However, we previously found that they do consider adding time to fix SonarQube issues.

Developers considered the process metrics, primarily the ADD (P1, P3-P8), OWN (P5, P7, P8) and EXP (P5-P8) metrics, to be good descriptions of their familiarity. However, none of these metrics is considered during estimates. When asked about code quality metrics P8, states,

"*We should think of it during estimates, but honestly we don't. We always consider quality in the back of our minds however, because we have to meet the project standards. Right now if we encounter bad code quality during the development of a task we often create an additional task. If the developer thinks he is unable to understand the code, due to for example poor quality, he creates an investigation task.*"

This creation of additional tasks, like an investigation task, is something only done in Project 4. In the other projects, developers will often revise their estimate or simply spend more time on the task than initially planned (resulting in inaccurate estimations). Because these investigation tasks will fall under the same PBI, they should be included in PBI effort estimations. However, if they are added at a later stage, they are a result of inaccurate estimations.

There is a clear consensus among the interviewees that code quality metrics are not considered in the estimations. The same reason holds for not inspecting files during estimates, namely that details at such a level do not add enough value. When we asked developers

what they think influences their estimation the most, P4 mentions,

"*The biggest impact is the size of the change we are going to introduce. I would measure the size by means of experience we had with the product in our team, as we are well versed with the existing code.*"

This is interesting because the developer mentions experience and size as related. While experience and size are mentioned by other developers (P1, P5-P8), they do not specifically mention that they are related.

> Developers use SonarQube to maintain code quality. No quality metrics are considered during effort estimates, but half of the developers add time to fix these issues later.

3.2 RQ2: How do code quality metrics relate to agile effort estimations?

We evaluate whether metrics can express the experience of the developer and the size of the task/PBI. This will tell us if effort estimations by developers can be expressed in empirical attributes. Furthermore, this will tell us if the experience the developer has with the code already links quality to estimates.

We investigate the relationship between effort estimates and the proposed metrics (section 2.5). We then consider a model in which these predictors are combined into a single model. In this section, we show the distribution of the estimates and the results for both regression techniques. Initially we assess the quality for effort estimations on a task. Next, we evaluate the relationship between quality and the sum of the effort of a PBI. The sum is all the estimated effort of the tasks that belong to the PBI.

3.2.1 Quality and task estimates

The dependent variable for regression has great influence over the results. Furthermore, the definition of a task will likely differ for every project. Hence, before considering the regression outputs we evaluate the distributions of the effort estimation for every project (Figure 4.1).

Project 1 tends to estimate tasks at eight hours, while other projects estimate many of the tasks under three hours. This indicates that the teams do not concur with the definition of a task. When we ask P1 during the interviews why Project 1 deviates so much from the rest, there was no reason given. Unlike other projects, they also follow a Fibonacci sequence when estimating tasks (Section 2.2.3). This can be seen by the great gap in Figure 4.1 for Project 1, in which there are nearly no tasks of the values 6, 7 or 9-12.

Most distributions are left skewed (there are many small tasks). This is something that is expected because the general idea behind a task is that one should be able to finish it within a day or less. Some projects, like Project 3 and Project 4 Team 2 appears to be left skewed.

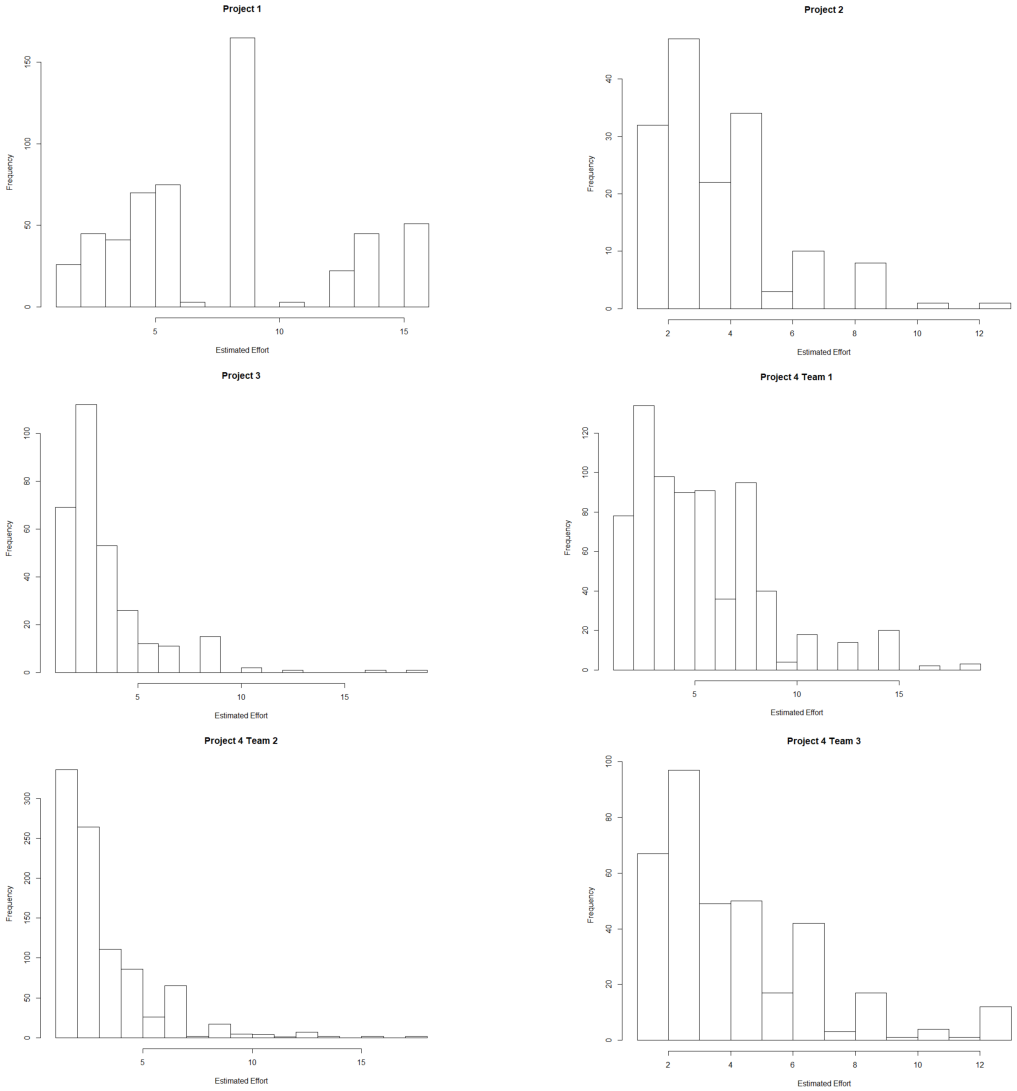


Figure 3.1: Distribution of estimated effort for the selected tasks for all projects

First, we will consider the technical code quality metrics. Before inputting the variables into the regression model, we evaluate how the variables are correlated with the estimated effort. We only report variables that are correlated more strongly than a weak ($r = 0.3$) positive or negative correlation. For the correlation to be statistical significant we require that $\rho < 0.05$.

For Project 1, the sum of the cyclomatic complexity of the classes (a metric calculated by Understand, not something we summed to reduce to task level) seems to have a weak correlation ($r = 0.37$) with the estimated effort. The sum of the cyclomatic complexity in a file intuitively would also correlate with the file size (which is indeed the case here). Hence, this should not be considered as just a measure of cyclomatic complexity. In Project 2,

the total methods in every file seems to have a correlation of $r = 0.35$. In team three, the number of lines in a file seems to correlate most strongly of all ($r = 0.4$). All the other metrics correlate lower than $r = 0.3$ for all the projects.

During the interviews, we found that ESDV developers focused on test coverage and cyclomatic complexity. Therefore, while manually selecting predictors, we choose to prioritize these metrics. There is, however, a point of criticism regarding these metrics because developers validate that they meet the ESDV standards for these metrics, e.g., < 15 cyclomatic complexity. They are not frequently considered to be poor. PCR does not involve a manual selection step and should give a different perspective. In Table 3.1, we can see the adjusted-R^2 values and significance results for both regression approaches for all the projects.

Table 3.1: Regression results for Technical Code Quality metrics

Project	PCR R2	ρ	Selection R2	ρ
Project 1	0.06	0.01	0.05	0.00
Project 2	0.09	0.02	0.12	0.00
Project 3	0.05	0.04	0.08	0.02
Project 4 Team 1	0.01	0.16	0.02	0.15
Project 4 Team 2	0.1	0.01	0.15	0.02
Project 4 Team 3	0.05	0.5	0.00	0.7

R^2 values close to 1 indicate that all dependent variables in the model can be explained by (a subset of) the given predictors. R^2 values close to zero, as is the case for code quality for the projects, indicate that only little of the data is explainable using the predictors. Among all the projects, the most common explainable variables are related to the number of methods reported, the ratio of the comment to code and the total lines of code present in the files. For the strongest results, Project 4 Team 2, the ratio of the comment to code had the biggest impact of all metrics. However, because the results are very weak, even these relationships are negligible.

When performing a simple Spearman correlation between the estimated effort and the technical code quality metrics, none of them are correlating above the $\rho = 0.3$ threshold. Therefore, the results indicate that the technical code quality metrics seem to have a minimal impact on the effort estimates of developers.

We apply the same technique, but for the process metrics and a combination of all metrics (Table 3.2 and Table 3.3). While selecting variables for the process metrics, we prioritize the ADD, OWN, EXP and COMM metrics. As the developers stated in the interviews, they perceive these metrics as the most defining process metrics. Because developers mentioned that they consider size an important factor during the estimation, we also prioritize the metrics related to the size of the change (LOC_ADD and MAX_ADD). In the combined version, we simply combine every selection and once again reduce highly correlated variables.

The ownership metric always seems to have one of the strongest correlation values and is the only metric to exceed $\rho = 0.3$. On average the correlation values for ownership are

in the range of 0.1 and 0.3. For project 1, 2 and 3, the total lines added to the task during the change correlate between a range of 0.3 and 0.45. For Projects 1 and 3, the total file and source changes perform similarly. The primary influence is frequently explainable by the lines and the total files one is going to modify during the task. These metrics are the reason Project 3 performs well. This seems to contradict what we would expect after the interviews, namely, that the size metrics would be an influential predictor.

Table 3.2: Regression results for process metrics

Project	PCR R2	ρ	Selection R2	ρ
Project 1	0.14	0.00	0.11	0.00
Project 2	0.12	0.06	0.15	0.04
Project 3	0.19	0.00	0.22	0.00
Project 4 Team 1	0.08	0.01	0.05	0.00
Project 4 Team 2	0.09	0.02	0.05	0.00
Project 4 Team 3	0.04	0.00	0.05	0.07

Surprisingly, the effect of the metrics is negligible. The primary influential metrics are related to age and the size of the work added. In Project 3, we observe the strongest metrics when the lines added metrics perform well. However, the overall results are weak. Next, we observe the combination of the technical code quality and process metrics.

Table 3.3: Regression results for combined metrics

Project	PCR R2	ρ	Selection R2	ρ
Project 1	0.2	0.00	0.16	0.00
Project 2	0.03	0.03	0.11	0.00
Project 3	0.23	0.02	0.26	0.00
Project 4 Team 1	0.09	0.07	0.04	0.00
Project 4 Team 2	0.12	0.04	0.06	0.00
Project 4 Team 3	0.1	0.1	0.07	0.07

Looking at the overall effect of the metrics on tasks, we observed that the primary impact of metrics that indicate the size or volume, either of the file itself (LOC, methods), or of the change in the file (LOC added, number of files). While cyclomatic complexity seemed to be an important driver in ESDV, we only see this influence in Project 1. The other metrics seemed to have a minor imperial measurable impact on the developers' effort estimations. We also observed that there are differences in the PCR and prediction selection regression methods, but the differences are not very strong.

3.2.2 Quality and PBI estimates

We consider an approach in which we do not calculate the effort and quality at a task level, but at a feature level (product backlog item), as discussed in Section 2.7. Table 3.4 shows the total input data for this method. We report the most significant R^2 values of the two regression methods. For the complete output, see Appendix D.

Table 3.4: PBI task sum relations

	T-Code Quality		Process		Combined	
Project	R2	ρ	R2	ρ	R2	ρ
Project 1	0.11	0.00	0.48	0.00	0.4	0.01
Project 2	0.07	0.16	0.23	0.00	0.35	0.00
Project 3	0.18	0.02	0.34	0.00	0.42	0.02
Project 4 Team 1	0.23	0.03	0.40	0.00	0.41	0.00
Project 4 Team 2	0.01	0.36	0.49	0.00	0.42	0.00
Project 4 Team 3	0.04	0.22	0.07	0.42	0.18	0.06

These results are stronger than the task results, especially for the process metrics. However, intuitively it makes sense that process metrics perform so well using this approach, especially those related to the size of the change. The size of a feature ranges from only a few hours to hundreds of hours. Figure 5.2 shows the distribution of the PBI effort of the two strongest performing process metric projects, project 1 and project 4 Team 2.

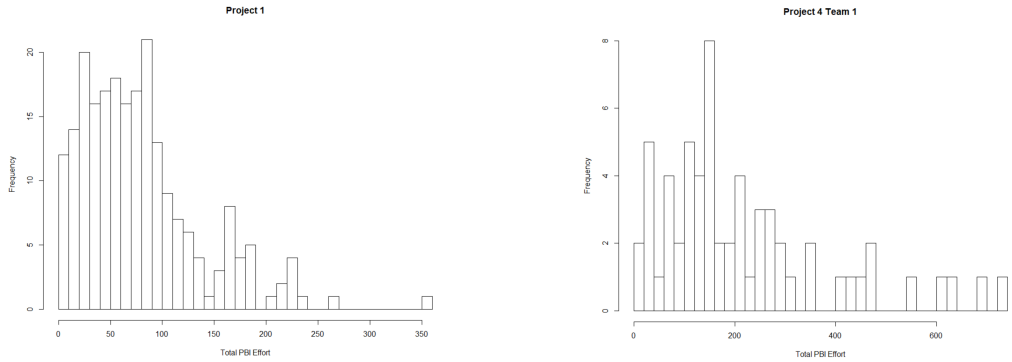


Figure 3.2: PBI velocity for Project 1 and Project 4 Team 1

While it is harder to estimate the exact number of hours, the accuracy is intuitively going to be higher than the task estimates, especially when one compares smoother estimate distribution (Figure 5.1 and Figure 5.2).

The most influential process metrics of all the projects is the lines of code added during the PBI, which informs us that developers can accurately predict the total size of the change for a greater time period. Other strong metrics are the maximum lines of code added during all changes to the PBI, further confirming this relationship. Moreover, the amount of

bugfixes the file was involved in and the age also show a minor relationship. While these effects are minimal, we see that legacy or bug prone code seem to influence developers, often resulting in higher estimations.

An interesting find is that the experience of project 4 Team 2 has a very strong effect, accounting for nearly all of the result. In Project 2 there seems to be a great influence by the commit count of the history of the developers. While the turnover is not that high, it certainly is the highest of all the considered teams. We speculate that this could be one of the reasons that process metrics perform better in Project 2, because new developers are less experienced with the code, they will likely estimate higher.

The results for the process metrics perform better than the combined metric in two cases. This can be due to two reasons. First, the selection procedure does not have to result in the most significant metrics. This is one reasons we also apply PCR regression. Second, in regression a larger amount of predictors does not have to generate a more significant result because the adjusted R2 values already compensate for a great part of this effect.

Code quality metrics seemed to have a small impact on Project 4 Team 1 and Project 3. The higher values are best explained by the mean of the average lines of code in the modified files. The sum of the tasks is something that should be comparable to the story points, because in a normal scenario all the tasks are estimated during the iteration planning. Therefore, consider the relationship among story points and the metrics (Table 3.5).

Table 3.5: PBI story point relations

	T-Code Quality		Process		Combined	
Project	R2	ρ	R2	ρ	R2	ρ
Project 1	0.03	0.59	0.35	0.00	0.07	0.08
Project 2	0.11	0.12	0.17	0.01	0.09	0.12
Project 3	0.02	0.29	0.25	0.18	0.33	0.01
Project 4 Team 1	0.28	0.12	0.32	0.00	0.32	0.00
Project 4 Team 2	0.02	0.23	0.34	0.00	0.37	0.00
Project 4 Team 3	0.01	0.29	0.01	0.35	0.14	0.4

We observe comparable results but, we notice that some results problems show statistical insignificance. The code quality metrics in particular seem to show problematic ρ values. The measures of the process metrics, while smaller than the previous approach, are still stronger than for a task. Like the summed hours tasks, this is an interesting observation. Because it indicates process metrics, and especially those related to the size of the change, have a stronger relationship with PBIs than with task estimations.

For tasks we observed very weak relationships among the metrics and effort estimations. We found stronger relationships among the process metrics for a PBI.

3.3 RQ3: When and how do developers encounter code quality during actual effort?

Before we look for an empirical relationship between code quality metrics and actual effort, we investigate how developers encounter and deal with code quality issues during development. This is done to determine what developers consider during agile effort estimations.

We simply ask the developers a series of questions (see Appendix C), to explore if they encounter poor quality during their development activities and how this affects their ability to implement new features. One of the three tasks that we ask developers about is a task of poor quality with an intermediate estimate (an estimation made after the task was started). This estimate should preferably be higher than the original estimation. This is an uncommon scenario, and hence was not available for all interviewees. We could only identify tasks with recent inaccurate estimation with poor quality for three developers.

All developers acknowledge they encounter scenarios in which they face code quality issues. They often mention legacy code, as P7 did.

"*We primarily have issues with legacy code that was developed before we started working on the project. Whenever we encounter legacy code of poor quality we try to include time to improve it in our planning for the current or next sprint.*"

Legacy code is often mentioned (P1,P4,P5,P7 and P8) as the most frequent scenario in which developers encounter poor quality. P1 and P4-6 acknowledge that when they work with bad quality legacy code, they take their time to refactor it. The other interviewees say it is highly dependent on the situation, whereas P7 mentions that if for example depends on the frequency of use of the legacy code. The frequency at which developers encounter legacy code is however limited according to all. Overall developers mention they do not encounter poor code quality frequently. None of the developers mentions that code quality is a big issue in their project.

We figured out if it was the unfamiliarity with the code that made the developers consider the legacy code of poor quality. As legacy code is something that is likely not recently maintained by the developers. P4 and P7 state that legacy code impacts their ability to estimate the task. Other interviewees say that they did not really consider whether it was legacy code or not during their estimations.

When we ask the interviewees about the length of the time lost on poor code quality, they consider it a difficult question to answer. P1 mentions 20% and P5 mentions 30%, but both emphasize that they are very uncertain about this number. However, all acknowledge it will likely result in more actual effort than estimated.

When we present the developers with the task none of them acknowledge it was due to poor code quality. P4 mentions:

"*The encountered task was more difficult than we initially estimated. After we completed our investigation task we noticed that there was more to the task and that it would take more time to complete.*"

Once again, we encountered the investigation task to initially inspect the work. All other developers give comparable reasons, they frequently (56%) made faulty estimates because it turned out the size of the work was more than estimated. P1 mentions:

"*If the task takes longer than I expected I will simply work longer on it. This can have an effect on the overall planning.*"

None of the interviewees addressed the poor quality (identified by our metrics) of the code that belongs to the tasks. When we presented the three developers with the tasks that are inaccurate and of poor quality, P1 acknowledged that he thought the quality of the code, among other things, was an issue during development. The other developers mention it is because they underestimated the total number of hours for the task, resulting in more time spent. P7 mentions:

"*I thought the implementation of the feature would take less time. However, during development I noticed there was more to it than was thought.*"

Why his task took longer is due to a highly technical reason, that required more effort than thought. One developer mentions that the actual over-estimate was not as great, while in the other cases, the actual calculation was not far off.

We ask developers about a scenario in which quality had an impact on their actual effort. P3 and P6 both address a scenario in which the project architecture was to blame, resulting in re-factors at an architecture level. P4,P5 and P7 address a scenario in which they did not fully understand the code and considered this to be a consequence of poor (documentation) quality. P1 and P8 mention a scenario where they could not understand the hierarchy of the code, losing sight of what is going on. When asked about how to handle code quality issues P9 says:

"*In case you have a lot of technical debt, like you encounter poor code quality during development or have a large amount of SonarQube issues or have a lot of bugs in the backlog, then you can add this as additional work during the estimation to the PBI during the sprint planning. However, this should also be considered during the day-to day planning.*"

While developers encounter quality issues in their every-day work, the majority mention legacy code as problematic. These cases are, however, isolated problems that developers only encounter on an irregular basis. Surprisingly, in the tasks that we considered to be of poor quality (according to the metrics), none of the developers actually address the fact that the code was of poor quality. Overall we found that developers do not encounter poor quality issues on a frequent basis.

Developers primarily encounter these poor quality scenarios on an irregular basis in legacy code. In all the 24 tasks of poor quality, none of the developers mentioned that quality was an issue.

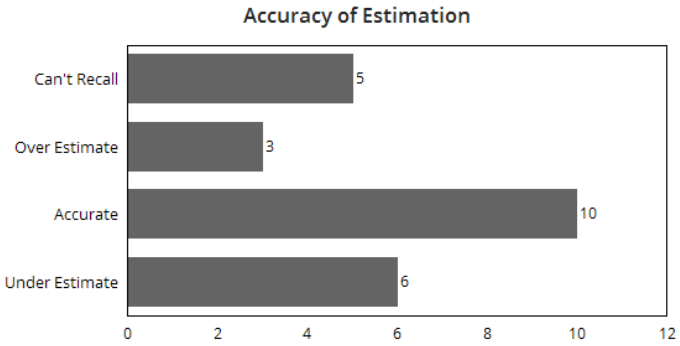
3.4 RQ4: How accurate are agile effort estimations?

To answer if poor quality does indeed result in developers spending more time to develop than anticipated, we empirically investigate the accuracy of the effort estimations. Furthermore, during the presented tasks, we asked the developers about the actual effort and how it compared to their estimate.

Accuracy of the estimates

We asked developers about the accuracy of their estimates by means of two methods. For every task we presented, we asked them how accurate the estimate actually was, and how accurate they consider their general estimates to be. Figure 3.3 shows the results for the tasks presented to the developers.

Figure 3.3: Accuracy of interview tasks



Three out of five developers could not recall the estimation accuracy of the cases. For instance, P7 had not been developing on the project for a while and hence was unable to recollect the exact hours spent on the tasks. While the data is not sufficient to consider the results significant, we can see that developers think they are accurate with their estimates. P4, however, mentions:

"*None of the estimated will be accurate. There will always be a room of 0.5-1h. Sometimes we complete less than the estimated hours. It depends on the person who picked it. Few people are well versed with the background services. If they pick it it will be completed before the estimated hours. If they are not very familiar with the code it will take longer.*"

This indicates that while the range of 0.5-1h is not that bad there will always be differences in precision of the estimations. Interestingly, we noticed another mention of the importance of experience from the developer.

Accuracy of empirical estimations

Before we observe the effect of the magnitude of relative error on quality, we first assess how the actual effort compares to the estimated effort (accuracy). Table 3.6 shows the total number of tasks that met the criteria from Section 2.9 for both approaches. We also see how

many of these tasks are considered development tasks; this is a result of intersecting these selected tasks and the tasks that serve as an input for research question one (Table 2.8).

Table 3.6: Tasks for research question 2

Project	One day technique	Dev. Tasks	Multiple-day technique	Dev. Tasks
Project 1	60	31	108	63
Project 2	180	14	290	22
Project 3	69	52	104	72
Project 4 Team 1	394	102	632	157
Project 4 Team 2	293	64	419	93
Project 4 Team 3	126	37	248	76

There is a large reduction in tasks for most projects. The reason is that some teams frequently drag a number of tasks to 'in progress' simultaneously. This simply means that developers sometimes work on multiple tasks at once. Next, we evaluate how these two techniques correlate with the estimated effort (Table 3.7). We choose a Spearman correlation because the results do not have to be linear. We also report the significance of the correlation.

Table 3.7: Correlation actual effort - est effort

Project	One day r	ρ	Mult. day r	ρ
Project 1	0.4	0.00	0.32	0.00
Project 2	0.17	0.02	0.3	0.00
Project 3	0.23	0.06	0.42	0.00
Project 4 Team 1	0.36	0.00	0.32	0.00
Project 4 Team 2	0.33	0.00	0.37	0.00
Project 4 Team 3	0.3	0.00	0.35	0.00

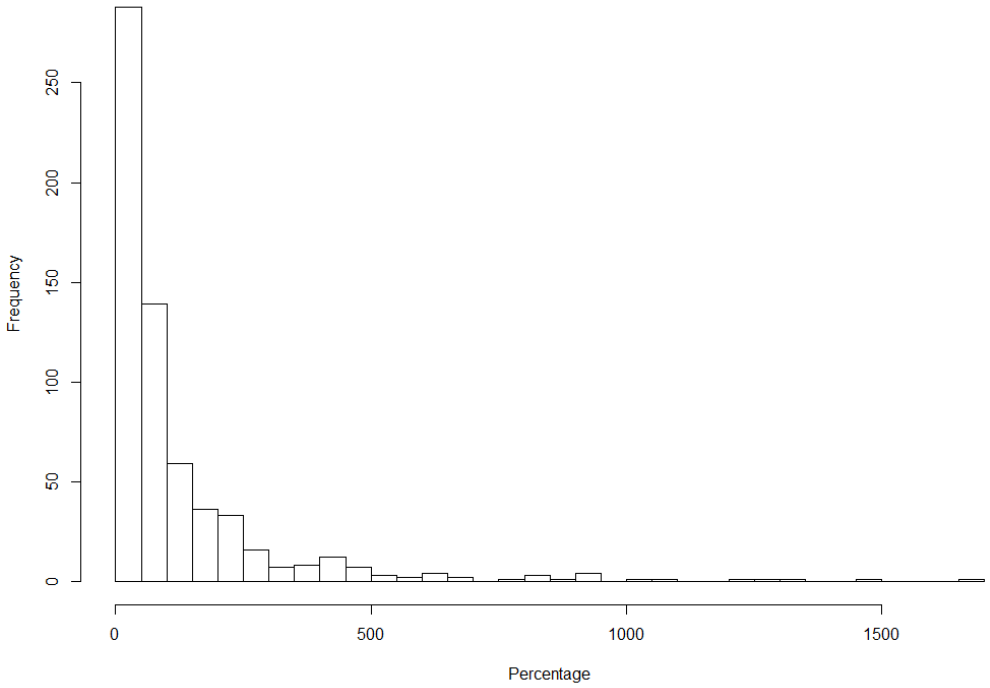
Developers could easily overshoot their estimated time for many reasons. Perfect estimation accuracy would result in perfect correlation. During the interviews developers sometimes mention an accuracy within 20-30%, something currently not reflected by the correlation values. To evaluate the accuracy we use the magnitude of relative error (MRE) [79], which expresses the difference between actual and estimated effort relative to the actual effort (formula 3.1). The MRE is commonly used to measure the accuracy of software effort estimations [81].

$$MRE = \frac{|\text{ actual effort- estimated effort }|}{\text{actual effort}} \tag{3.1}$$

This should help to identify outliers in the data. Consider the distribution for the largest dataset, Project 4 Team 1, in Figure 3.4.

One case requires close to 2000% more time than the estimated time. There appears to be several such tasks in every project. This behavior seems to be either errors (due to

Figure 3.4: Distribution of the absolute MRE for Project 4 Team 1



our applied methods) or has an unidentified reason, and has a great influence on both the regression and the correlation.

We manually inspect all peculiar cases (from Table 3.6) and remove unrealistic ones. During manual inspection we investigate (if we have access) the agenda of the developer on the date. When we observe something that was non development related we immediately remove the task from the set of considered tasks. After that we manually inspect the contributions made and if the effort does not justify the contribution we exclude it.

This has a positive impact on the correlation, changing the mean values from the previous r = 0.3 to about r = 0.5. Project 2 in particular shows significant improvements, from 0.3 to a correlation of 0.6. This is due to some very extreme differences between the actual and estimated effort (one being 4000% more).

As mentioned in Section 2, Project 1 very recently started using a 'completed work' field, that describes the amount of time, in hours, that it took to complete a task. We treat the completed work field equally to that of the actual work field, because it should describe the same amount of hours worked on. We call this case *Project 1 CW* from here on. This field would be a perfect evaluation of the performance of our 'actual effort' method. However, there are currently only 8 out of the 108 tasks from the multiple day technique that overlap. We consider this an insufficient amount to be significant for evaluation. The reason is that

this field has been in use for less than three months, and the previously mentioned behavior of Project 1, in which they simultaneously put many tasks to 'in progress'.

Let us consider all the tasks of Project 1 that have both a completed work field and an effort estimate, a total of 291 tasks. The Spearman correlation between these two variables is 0.65. One could once again question if this should be set as a goal, but it should give an indication about the accuracy of the estimate in general, indicating that a very strong correlation does not have to represent reality. Out of these 291 tasks 39 are development tasks. These tasks serve as the input for regression. Other than correlation we can calculate the accuracy of the effort estimations, or the estimation error. By taking the mean of all the MRE values (MMRE), or the approximation error, for all the considered tasks in a project (Table 3.8).

Table 3.8: The MMRE for all projects

Project	**One Day**	**Mult Day**
Project 1	130%	97%
Project 1 CW	-	91%
Project 2	78%	81%
Project 3	85%	75%
Project 4 Team 1	67%	63%
Project 4 Team 2	66%	65%
Project 4 Team 3	69%	66%

The level of inaccuracy seems a little higher for project 1 than other work, in which teams report an MMRE of 28-90% [81]. Only Project 1 seems to be inconsistent with other literature. What is surprising is that Project 1, Project 1 CW, Project 2, Project 3 and Project 4 Team 1 under-estimate their estimates while Project 2 and Project 4 Team 2 and Team 3 over-estimate their estimations. We observe no large difference between the completed work field and the calculated method for Project 1, indicating that the method perform accurately. We also observe no large difference between the multiple day and the one day technique for any project.

We found that the estimation error (MMRE) ranges from 63-130%. The interviews indicated that developers considered more than half of their estimations to be accurate in a range of $\pm 1h$.

3.5 RQ5: What is the influence of code quality metrics on the accuracy of agile effort estimations?

We will assess the relationship between the code quality metrics and the mean magnitude of relative error (MMRE) assessed in the previous question. If we find a (positive) relationship, it would indicate that the code quality metrics have an effect on the extended development time. We apply the same technique as in Section 3.1, also applying the same metric selection procedure based on the results obtained in the interviews. The input consists of the same tasks selected during research question four. The results for the code quality metrics can be seen in Table 3.9.

Table 3.9: Technical Code quality metrics with MRE

	One Day Technique				Mult. Day Technique			
	PCR	ρ	SEL	ρ	PCR	ρ	SEL	ρ
Project 1	0.1	0.12	0.14	0.09	0.12	0.03	0.14	0.00
Project 1 CW	-	-	-	-	0.15	0.06	0.16	0.00
Project 2	0.00	0.87	0.03	0.67	0.01	0.83	0.06	0.74
Project 3	0.03	0.45	0.06	0.87	0.1	0.82	0.02	0.7
Project 4 Team 1	0.00	0.27	0.03	0.48	0.00	0.82	0.01	0.71
Project 4 Team 2	0.11	0.09	0.07	0.14	0.06	0.76	0.1	0.81
Project 4 Team 3	0.03	0.16	0.05	0.27	0.00	0.12	0.00	0.37

Many results seem statistically insignificant, meaning they exceed the threshold of $\rho > 0.05$. This can be explained by the large reduction in data with the applied methods. The biggest impact of code quality seems to be in Project 1. The other projects seem to show very low values and seem to be of no significance. It is interesting to see that the results of the completed work field and the applied method are very similar, hinting that the devised method to calculate actual effort shows promise. The metrics with the highest impact for Project 1 are the test coverage, coupled classes, inheritance tree, declared methods and the total amount of semicolons in a file.

Table 3.10: Process metrics for MRE

	One day Technique				Mult. day Technique			
	PCR	ρ	SEL	ρ	PCR	ρ	SEL	ρ
Project 1	0.18	0.22	0.16	0.32	0.19	0.04	0.21	0.03
Project 1 CW	-	-	-	-	0.22	0.03	0.17	0.04
Project 2	0.11	0.55	0.22	0.71	0.06	0.87	0.07	0.79
Project 3	0.12	0.49	0.08	0.35	0.04	0.69	0.09	0.61
Project 4 Team 1	0.17	0.03	0.19	0.04	0.17	0.04	0.14	0.06
Project 4 Team 2	0.13	0.06	0.19	0.06	0.07	0.30	0.17	0.29
Project 4 Team 3	0.08	0.32	0.16	0.43	0.09	0.26	0.06	0.21

Table 3.10 clearly shows that most of the results do not support our hypothesis. When looking at process metrics, a very common metric to have a decent impact on the model is 'age', meaning that older files seem harder to maintain than recent ones. Legacy code was also considered very influential during the interviews. Hence we reason that this relationship exists because legacy code could be indicated by the age metric. Furthermore, the metrics related to the lines added and removed seem to have the biggest impact. In addition, we see an effect from the total number of files that was changed.

Table 3.11: Combined metrics for MRE

	One day Technique				Mult. day Technique			
	PCR	ρ	SEL	ρ	PCR	ρ	SEL	ρ
Project 1	0.22	0.25	0.17	0.22	0.23	0.04	0.28	0.01
Project 1 CW	-	-	-	-	0.19	0.09	0.21	0.07
Project 2	0.00	0.83	0.35	0.76	0.00	0.9	0.1	0.76
Project 3	0.04	0.76	0.07	0.65	0.02	0.87	0.05	0.83
Project 4 Team 1	0.07	0.00	0.12	0.01	0.1	0.03	0.06	0.06
Project 4 Team 2	0.21	0.06	0.2	0.02	0.06	0.32	0.17	0.42
Project 4 Team 3	0.00	0.4	0.03	0.45	0.03	0.51	0.07	0.63

We see a weak (0.2-0.3) relationship between the quality and a prolonged development time for Project 1. We speculate that this stronger relationship in project exists because they do not seem to use any quality tooling. They use no tooling measure the code quality metrics maintain a certain standard; hence, there is more fluctuation and impact by the code quality. The other projects, however, acknowledge they have been using quality maintenance tooling for between four to twelve months.

We found weak relationships among the estimation accuracy and the metrics. We find the strongest (R^2 0.3) results for Project 1, the only project that does not use any quality tools.

Chapter 4

Discussion

4.1 Metric selection

It is common for software projects to collect an excessive amount of metrics [82]. During the reduction of the metrics in the predictor selection procedure we saw that many metrics are very strongly correlated (Table 2.7 and Table 2.8). This had a significant impact on the results, as without this procedure the results would have been a lot stronger. While *Understand* can calculate a large amount of technical code quality metrics (Appendix A), their similarities are astonishing.

The fact that a large amount of metrics did not produce better or more significant results has been found in previous work [82]. During selection we also noticed that, like previous work [17], the metrics often form a pattern of: volume, complexity, sizing and coupling. We found that the developers in Shell often optimize for a small set of metrics. While not all of the groups of metrics are covered, this does indicate that pursuing only a small amount of carefully selected metrics can have great implications. We do however recommend pursuing more metrics than just testing and cyclomatic complexity, especially because size and volume related metrics were the most influential metrics in most of our results.

Table 2.8 shows that, while combining the metrics, the correlation among the three groups is not strong. This indicates that the three groups of metrics are sufficiently different from each other that it seemed to make sense to consider them separately.

4.2 Code quality and effort estimations

In RQ1 (Section 3.1), developers mention they did consider their experience and the size of the change important influences during effort estimations. The importance of size and experience for effort estimations are also mentioned in previous literature [67, 31]. Some developers said their ability to estimate the size becomes more accurate with experience. Developers acknowledged they are aware of technical code quality, but none consider it in depth by looking at static analyses tools. However, half the developers add a static amount of time to their tasks simply to resolve any problems generated by SonarQube. So, we do observe impact on the effort estimations, but it is primarily a static amount of time.

In RQ2 (Section 3.2) we attempted to quantify a relationship between effort estimations, size and experience. We use size related process metrics to quantify size, the process metrics related to experience to quantify experience, and technical code quality metrics to measure the maintainability of the code. During the interviews we found that the size of the change and experience are considered during effort estimations. Therefore we suspected that process metrics would perform well.

We found weak relationships among all of the metrics and task effort estimations. Evidently, there is more to size than the additions made in terms of code volume, but these metrics did show such minor relationships for tasks. However, studies that focus on size metrics comparable to ours (lines of code related) for agile effort predictions did not have high prediction accuracy [10] either. Hence we provide further evidence that size measured by lines has only minor implications for agile effort prediction.

We observed similar results for process metrics and technical code quality metrics for a task. Our expectation was to find at least some relationship among effort estimates and technical code quality metrics, as we hypothesized bad experience and poor code quality would result in higher estimates. However, during the interviews we found that developers do not seem to incorporate technical code quality during their estimates. Some teams mention they assign a static amount of hours to fix these issues, which should result in an effort increase independent of the quality of the task.

We suspect that we observe weak overall relationships because there are many more factors to the estimations than the metrics we considered. As an example, Soh et al. [78] find that developers spent the majority of their time exploring what files they have to modify during maintenance related tasks. Secondly, in Figure 3.2.1 clearly shows that estimations are frequently low (1-3 hours) and inaccurate (Table 3.8). Some developers even mention during the interviews that their estimates are just a guess, and have very little outside influence. We question if this left skewed set of inaccurate effort estimations serve as a proper dependent variable for regression.

We further investigated the relationships among the effort of the tasks that are part of a PBI. We found stronger relations among change size-related metrics, and in two projects, stronger relationships for process metrics. These results hold for both story points and the sum of the effort of the tasks. This is surprising, because it hints that story point (PBI) estimations are more affected by or more accurately predict the size than the estimates assigned to tasks.

Project 2, the project with the highest turnover, performed worst in estimating the size. We speculate that this has something to do with developers' statements, i.e., that the ability to predict the size of the change depends on the experience of the developers.

We observed overall stronger performance for the PBI process metrics compared to the tasks. In particular, metrics related to the experience, like the experience metric and the commit count of the developers, seemed to have some effect. We reason that stronger relations exists because working on legacy code requires more effort. Something also mentioned by developers. For a task we looked at a single developer, while for a PBI we looked at the experience of the developer(s) who worked on that PBI (often multiple). When we consider process metrics for a task it only describes the experience of that developer, but the experience of the multiple developers involved in the PBI might be a better representation

of experience.

Hence, code quality seems to hold a stronger influence on PBI estimates. However, technical code quality was weak for both tasks and PBIs. The method developers seem to use to maintain technical code quality is by adding a static amount time to every task to fix generated quality violations after development. We expected stronger results for the process metrics, especially because developers mention they consider these factors important during their effort estimations, but overall the impact of code quality and the metrics on effort estimations was minor.

4.3 The impact of quality on actual effort

In RQ3 (Section 3.3) we observed rather high levels of inaccuracy (Table 3.8) for effort estimations, but such levels of inaccuracy are observed in prior work as well [81]. We did observe certain extreme outliers to the actual effort calculation, confirming (Section 2.12) that the method imposes limitations on the results. However, we observed striking similarities between the actual effort results and the completed work field from Project 1. Furthermore, we observed that, during the interviews, developers thought their estimations were accurate (56% accurate, 33% under estimate 21% over estimate). Some developers mention 20% to 30% of their estimations are inaccurate.

During the interviews *in RQ4* (Section 3.4) we presented the developers with tasks that contained poor technical code quality according to *Understand*. None of the developers actually acknowledged the fact that they considered these tasks of poor quality. They all mentioned specific technical reasons or simply that it took overall more time than estimated. This hints that the definition of poor according to these metrics might not be as influential during actual effort. Furthermore, developers mentioned that in the large majority of their time they consider the code they work on of good quality.

Developers seemed to frequently encounter code quality issues in legacy code. This indicated that legacy code, while encountered irregularly, seemed to be have a big impact on actual effort. During the empirical evaluation between the accuracy and the code quality metrics we found that the age metrics performs rather well. We reason estimation accuracy can be improved for these legacy code scenarios. Developers could take prior considerations about the state of the code, by considering process and other age related metrics. This would open the possibility to allocate time to improve or refactor the legacy code.

In RQ5 (Section 3.5) there seemed to be low relationships among code quality metrics and the estimation accuracy. What we did observe is that the strongest results are those of Project 1. This is worth mentioning, because Project 1 is the only project that does not use any tools to maintain quality. We can only speculate that this is the reason why they have the strongest relationship. If the developers do not maintain the overall code quality via static analysis tooling, they could face maintainability issues during development. These unforeseen maintainability issues could require additional time, resulting in higher inaccuracies. This hints that pursuing code quality metrics does improve maintainability [17].

Even though Project 1 does not pursue quality metrics, the team members review their code regularly. Code reviewing may improve software quality, style and properties like

shared ownership [59, 16]. During the interviews we find that all of the developers acknowledge they encounter poor quality code on an irregular basis or in legacy code. The pursuit of review coverage, SonarQube and the adoption of DevOps could have resulted in a very good quality state for the chosen repositories. Combined with the additional effort already assigned by a subset of developers for quality assurance, quality issues could be a rare occasion, something developers acknowledge during the interviews. On rare occasions, we would indeed observe a negligible effect.

Chapter 5

Related Work

(Agile) Effort estimations

There are often multiple approaches for effort estimations. There are several methods to estimate the total cost of the project [13, 18, 25, 49]. The scope and end goal of these models are to predict the cost or effort required to complete the whole project. The input of these models consists primarily of methods to measure software properties, such as size, complexity or the analysis of productivity [27, 70].

We question the implications of these methods in agile software development because as these methods require careful planning and considerations before starting the project, something not endorsed in agile software development. There is however plethora of research that tries to perform effort estimations in an agile setting [81]. This research frequently tends to only focus on the planning process, such as planning procedures performed during agile software development, and not on the relationship with the quality of the code.

There are several iteration planning techniques in agile software development. Frequently used techniques include: planning poker [45], usese case points [66], expert judgment [9], linear regression and neural networks [81]. Abrahamsson et al. [9] use a set of metrics derived from user stories, a prominent method to define requirements in agile [30], to predict the implementation effort required. They extract measures such as number of characters, words, but also the frequency of certain keywords like 'test' and 'report' from the user stories. Using linear regression they find rather poor accuracy results and hence limitation implications for such methods.

Popli and Chauhan [67] mention four common methods to perform agile effort estimation for the complete software development life-cycle. The learning-orientated approach, which use collective learning techniques from previous estimations, and experience and knowledge from managers, creates the planning. The expertise-based approach, uses an expert to defines the long-term planning for the project. A regression method, where the model consist of developing regression equations to make estimations and the bottom up approach, where all components are separately estimated and the results are aggregated to produce an overall estimate. However, none of these methods seem to devote much attention to scenarios where projects have to work on existing code. Or the everyday life of a software developer.

Previous agile studies primarily use size or cost metrics as predictor for effort [29]. To the best of our knowledge no projects use technical code quality or process metrics as a predictor for agile effort.

Metrics and Software Maintainability

There are different methods studies define quality in agile. Some studies measure it by means of customer satisfaction [8]. A large number of the studies that revolve around the quality maintenance in agile evaluate the effect of engineering best practices, such as pair programming or test driven development. These techniques show great quality improvements throughout the majority of the studies [74]. While we certainly agree they are good practise, they are not adopted by all agile projects, as they require discipline and investment.

Sjøberg et al. [77] try to quantify the effect of Code Smells on maintenance effort. They hire six developers to perform a controlled experiment. The developers are asked to perform maintenance work on code with different flaws in code smells. They use regression to quantify the relationship among the code smells and effort spend on maintaining the files. They seem to find no significant code smell metrics that associate with an increase in effort. The file size and the number of changes seem to be the primary explainable variable.

Baggen et al. [17] provide an overview how the Software Improvement Group [1] performs code analysis and quality consulting. Their primary incentive is reduce software maintainability. They follow the ISO/IEC 9126 definition of maintainability, a standard that has been replaced after the publishment of the article by ISO 25010 [2]. With respect to maintainability they come up with six source code properties that can be expressed in metrics: Volume, Redundancy, Unit size, Complexity, Unit interface sizing and Coupling [46]. Nugroho et al. [64] create an empirical model of technical debt and interest. They use SIG maintainability properties and use these qualities to measure technical debt.

Yuming and Baowen [84] investigate if a set of 15 design metrics can predict the maintainability of open source software. They quantify maintainability using a maintainability index, a formula more frequently used, which is made up of three metrics: Halstead's Volume [43], cyclomatic complexity and lines of code. They find that size and complexity metrics are strongly related to the quantified maintainability in open source projects. Cohesion and coupling on the other hand show no significance. They succeed in fairly accurate maintenance effort predictions (MRE $<$ 30%). However, Sjøberg et al. [76] find that a maintainability index are mutually inconsistent, nor is there much evidence that regarding the software maintainability index [71]. They come to a comparable conclusion as their previously discussed work, where they indicate that size metrics (class size, method size and file size) are underrated by the software engineering community as a measure of maintainability.

Oman and Hagemeister [65] propose a large set of metrics to measure the maintainability of software projects. These source code metrics are placed in a tree with three main branches: Maturity attributes, source code and supporting documentation. There are a lot

[1] https://www.sig.eu/

[2] http://iso25000.com/index.php/en/iso-25000-standards/iso-25010

of inspiring metrics found in their work, however none of them revolve around the experience of the developer (process metrics), while other research points out that experienced developers can reduce maintenance effort [51, 52]

One can see there are different approaches to define maintainability and maintainability effort. Shen et al. [75] empirically investigate the relation among maintainability and coupling metrics. They quantitatively define maintenance by the amount of software changes made to a certain file.

Capra et al. [27]. perform an empirical study that attempts to quantify the relationship among software design quality, development effort and governance in open source projects. They hypothesize that, as projects approach the OS end of continuum, governance becomes less formal. Their results indicate that software design quality, mainly measured by coupling and inheritance, does not have to increase development effort, but do show to be good variables to implement more open governance.

Frequently research tries to investigate the relationship among software design quality and maintenance development effort [21]. Design quality is frequently expressed by means of metrics, such as coupling or cohesion [33]. Alshayeb et al. [14] research if object oriented metrics can predict size of effort. They define effort by the total lines of source code added, changed or deleted. They also investigate if the object orientated metrics can predict the maintenance effort (in hours). They look for relationships using multiple linear regression. It seems the metrics are unable to predict maintenance effort, but do find that the object oriented metrics are able to predict the upcoming source lines added, deleted or changed. Their methodology to relate object oriented metrics to quality is not uncommon [26].

Soh et al. [78] empirically investigate how developers spend their task maintenance effort. For a large amount of users they crawl through their open source projects to look for relations. They measure the complexity of a task using the cyclomatic complexity metric and define effort by the total amount of changes a developer made in a patch. They find no correlation between the complexity of a task and the total amount of effort spent. Instead they find that developers spend the majority of their time on exploring what files they have to modify. They also find that the developers experience does not reduce maintenance effort, but observe, that as a program evolves, developers tend to perform tasks in sections of the code they are unfamiliar with.

Rahman and Devanbu [69] evaluate how process and code quality metrics serve in defect prediction models. Their results suggest that process metrics are better for defect prediction than code quality metrics. They find that code metrics may not evolve with the changing distribution of defects. Their combination of both metrics cover a great amount of the covered influential metrics. Hence, we choose to use the similar tooling and process metrics they use.

Research that mines software repositories often defines quality by means of post-release defects or failures[59, 22, 38, 58, 63, 68, 50, 28]. As an example Bird et al. [23] try to examine the effect of ownership metrics (a subset of our process metrics) and software quality. They define quality as the number of post release failures. They use correlations to evaluate the relations and conclude that high values of ownership and major (section 2.5.2) are associated with less defects.

McIntosh et al. [60] explore the empirical relationship between modern code review

and software quality. They build regression models for three large software systems that explain the incidence of post-release defects on the systems. They find strong empirical evidence to support their claim. Projects that have high code review coverage, participation and expertise will lead to less defect-prone software and hence higher quality. They provide a methodology for multiple linear regression, which we apply in section 2.10.2.

Chapter 6

Conclusions

During agile software development developers often plan in iterations. To set the goals for these iterations developers will estimate the size of the features they want to realize. If the developers have to maintain existing code, the quality of that code will likely impact the effort required to realize those goals. In this study we investigated what the effects of code quality are on agile effort estimations, and if prior knowledge of the code quality would have a positive benefit on the estimation accuracy.

To this aim we took a mixed method approach where we interviewed nine developers and mined the repositories of four large software projects. To collect and analyze empirical data we developed a tool to mine repositories and visualize data from Microsoft Visual Studio Team Services. All the data and interviewees originated from ESDV, an IT department in Shell.

We found that developers rarely consider technical code quality during agile effort estimations. Quality considerations were not integrated in the estimation process. Half of the developers add a static amount of time to their estimations to fix any quality issues found by static analysis tooling. Developers mentioned that the key considerations during their effort estimations are the size of the change and their experience with the code.

Next, we turned to quantifying the relationship between effort estimations and technical code quality, experience and size. We found that there was a negligible relation among any of the metrics and task effort estimates. For product backlog items the relationship was stronger for both process metrics and especially change metrics. This indicates that story points are more driven or related to size than (task) estimates. The ability to estimate the size seemed to scale as the team became more experienced.

Our results show that considerations of code quality metrics prior to the effort estimation would have had a minimal impact on the estimation accuracy. The team that suffered the most from estimation inaccuracies due to code quality was the only team that did not use any static analysis tooling, thus hinting that the use of static analysis tooling may eventually lead to improved maintainability of the code.

The effect of code quality on effort estimations did not seem negligible, but more of an irregular occurrence. For example, developers mention that they frequently face code quality issues while working on legacy code. Prior identification of these scenarios could be beneficial for the developers. However, our study indicates that in general scenarios the

effect of code quality is of minor importance on agile effort estimations. With this thesis we make the following main contributions:

- An analysis on the effect of code quality on agile effort estimations. Our results give agile developers an understanding of the importance of code quality considerations during their iteration planning;
- An understanding of how well agile effort estimations can be quantified. This will help researchers understand how feasible it is to quantify this relationship;
- A perspective from industry on how developers maintain their code quality and when they encounter code quality issues. This will help researchers and agile developers with the first steps to identify and overcome these quality issues.

Bibliography

[1] Designite faq containing metric tresholds. http://www.designite-tools.com/faq/. Accessed: 2017-03-13.

[2] ISO software quality standards. https://www.iso.org/standard/35733.html. Accessed: 2017-08-28.

[3] Scrum guide. http://www.scrumguides.org/scrum-guide.html. Accessed: 2017-09-01.

[4] Shell global ranking. http://beta.fortune.com/global500/royal-dutch-shell-5. Accessed: 2017-03-13.

[5] Sonarqube metric definition. https://docs.sonarqube.org/display/SONAR/Metric+Definitions. Accessed: 2017-07-13.

[6] Using pcr in r. http://www2.imm.dtu.dk/courses/27411/eNotepdfs/eNote4-PCRinR.pdf. Accessed: 2017-07-10.

[7] Visual studio team services work item structure. https://www.visualstudio.com/en-us/docs/work/guidance/scrum-process-workflow.

[8] Noura Abbas, Andrew M Gravell, and Gary B Wills. The impact of organization, project and governance variables on software quality and project success. In *AGILE Conference, 2010*, pages 77–86. IEEE, 2010.

[9] Pekka Abrahamsson, Ilenia Fronza, Raimund Moser, Jelena Vlasenko, and Witold Pedrycz. Predicting development effort from user stories. In *Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on*, pages 400–403. IEEE, 2011.

[10] Pekka Abrahamsson, Raimund Moser, Witold Pedrycz, Alberto Sillitti, and Giancarlo Succi. Effort prediction in iterative software development processes–incremental versus global prediction models. In *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, pages 344–353. IEEE, 2007.

[11] O Ege Adali, N Alpay Karagöz, Zeynep Gürel, Touseef Tahir, and Cigdem Gencel. Software test effort estimation: State of the art in turkish software industry. In *Software Engineering and Advanced Applications (SEAA), 2017 43rd Euromicro Conference on*, pages 412–420. IEEE, 2017.

[12] Manish Agrawal and Kaushal Chari. Software effort, quality, and cycle time: A study of cmm level 5 projects. *IEEE Transactions on software engineering*, 33(3), 2007.

[13] Yunsik Ahn, Jungseok Suh, Seungryeol Kim, and Hyunsoo Kim. The software maintenance project effort estimation model based on function points. *Journal of Software: Evolution and Process*, 15(2):71–85, 2003.

[14] Mohammad Alshayeb and Wei Li. An empirical validation of object-oriented metrics in two different iterative software processes. *IEEE Transactions on software engineering*, 29(11):1043–1049, 2003.

[15] Maurício Aniche, Christoph Treude, Andy Zaidman, Arie van Deursen, and Marco Aurélio Gerosa. Satt: Tailoring code metric thresholds for different software architectures. In *Source Code Analysis and Manipulation (SCAM), 2016 IEEE 16th International Working Conference on*, pages 41–50. IEEE, 2016.

[16] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 international conference on software engineering*, pages 712–721. IEEE Press, 2013.

[17] Robert Baggen, José Pedro Correia, Katrin Schill, and Joost Visser. Standardized code quality benchmarking for improving software maintainability. *Software Quality Journal*, 20(2):287–307, 2012.

[18] Victor Basili, Lionel Briand, Steven Condon, Yong-Mi Kim, Walcélio L Melo, and Jon D Valett. Understanding and predicting the process of software maintenance release. In *Proceedings of the 18th international conference on Software engineering*, pages 464–474. IEEE Computer Society, 1996.

[19] JP Bernard, J Sahel, M Giovannini, and H Sarles. Pancreas divisum is a probable cause of acute pancreatitis: a report of 137 cases. *Pancreas*, 5(3):248–254, 1990.

[20] Sonali Bhasin. Quality assurance in agile: A study towards achieving excellence. In *AGILE India (AGILE INDIA), 2012*, pages 64–67. IEEE, 2012.

[21] Aaron B Binkley and Stephen R Schach. Validation of the coupling dependency metric as a predictor of run-time failures and maintenance measures. In *Proceedings of the 20th international conference on Software engineering*, pages 452–455. IEEE Computer Society, 1998.

[22] Christian Bird, Nachiappan Nagappan, Premkumar Devanbu, Harald Gall, and Brendan Murphy. Does distributed development affect software quality?: an empirical case study of windows vista. *Communications of the ACM*, 52(8):85–93, 2009.

[23] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. Don't touch my code!: examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 4–14. ACM, 2011.

[24] B Boehm and K Sullivan. Software economics: status and prospects. *Information and Software Technology*, 41(14):937–946, 1999.

[25] Barry W Boehm, Ray Madachy, Bert Steece, et al. *Software cost estimation with Cocomo II with Cdrom*. Prentice Hall PTR, 2000.

[26] Lionel C Briand, Jürgen Wüst, and Hakim Lounis. Replicated case studies for investigating quality factors in object-oriented designs. *Empirical software engineering*, 6(1):11–58, 2001.

[27] Eugenio Capra, Chiara Francalanci, and Francesco Merlo. An empirical study on the relationship between software design quality, development effort and governance in open source projects. *IEEE Transactions on Software Engineering*, 34(6):765–782, 2008.

[28] Cagatay Catal and Banu Diri. A systematic review of software fault prediction studies. *Expert systems with applications*, 36(4):7346–7354, 2009.

[29] Evita Coelho and Anirban Basu. Effort estimation in agile software development using story points. *International Journal of Applied Information Systems (IJAIS)*, 3(7), 2012.

[30] Mike Cohn. *User stories applied: For agile software development*. Addison-Wesley Professional, 2004.

[31] Mike Cohn. *Agile estimating and planning*. Pearson Education, 2005.

[32] Kieran Conboy and Brian Fitzgerald. Method and developer characteristics for effective agile method tailoring: A study of xp expert opinion. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(1):2, 2010.

[33] David P Darcy, Chris F Kemerer, Sandra A Slaughter, and James E Tomayko. The structural complexity of software an experimental test. *IEEE Transactions on software engineering*, 31(11):982–995, 2005.

[34] Karel Dejaeger, Wouter Verbeke, David Martens, and Bart Baesens. Data mining techniques for software effort estimation: a comparative study. *IEEE transactions on software engineering*, 38(2):375–397, 2012.

[35] Tore Dybå and Torgeir Dingsøyr. Empirical studies of agile software development: A systematic review. *Information and software technology*, 50(9):833–859, 2008.

[36] Bradley Efron. *The jackknife, the bootstrap and other resampling plans*. SIAM, 1982.

[37] Bradley Efron and Robert Tibshirani. Bootstrap methods for standard errors, confidence intervals, and other measures of statistical accuracy. *Statistical science*, pages 54–75, 1986.

[38] Khaled El Emam, Saïda Benlarbi, Nishith Goel, and Shesh N. Rai. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Transactions on Software Engineering*, 27(7):630–650, 2001.

[39] Bent Flyvbjerg. Five misunderstandings about case-study research. *Qualitative inquiry*, 12(2):219–245, 2006.

[40] Takafumi Fukushima, Yasutaka Kamei, Shane McIntosh, Kazuhiro Yamashita, and Naoyasu Ubayashi. An empirical study of just-in-time defect prediction using cross-project models. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 172–181. ACM, 2014.

[41] Des Greer and Yann Hamon. Agile software development. *Software: Practice and Experience*, 41(9):943–944, 2011.

[42] James Grenning. Planning poker or how to avoid analysis paralysis while release planning. *Hawthorn Woods: Renaissance Software Consulting*, 3, 2002.

[43] Maurice Howard Halstead. *Elements of software science*, volume 7. Elsevier New York, 1977.

[44] Donald E Harter, Mayuram S Krishnan, and Sandra A Slaughter. Effects of process maturity on quality, cycle time, and effort in software product development. *Management Science*, 46(4):451–466, 2000.

[45] Nils Christian Haugen. An empirical study of using planning poker for user story estimation. In *Agile Conference, 2006*, pages 9–pp. IEEE, 2006.

[46] Ilja Heitlager, Tobias Kuipers, and Joost Visser. A practical model for measuring maintainability. In *Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the*, pages 30–39. IEEE, 2007.

[47] Siw Elisabeth Hove and Bente Anda. Experiences from conducting semi-structured interviews in empirical software engineering research. In *Software metrics, 2005. 11th ieee international symposium*, pages 10–pp. IEEE, 2005.

[48] Michael Httermann. *DevOps for developers*. Apress, 2012.

[49] Chris F Kemerer. An empirical validation of software cost estimation models. *Communications of the ACM*, 30(5):416–429, 1987.

[50] Foutse Khomh, Tejinder Dhaliwal, Ying Zou, and Bram Adams. Do faster releases improve software quality?: an empirical case study of mozilla firefox. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, pages 179–188. IEEE Press, 2012.

[51] Barbara A Kitchenham, Guilherme H Travassos, Anneliese Von Mayrhauser, Frank Niessink, Norman F Schneidewind, Janice Singer, Shingo Takada, Risto Vehvilainen, and Hongji Yang. Towards an ontology of software maintenance. *Journal of Software Maintenance*, 11(6):365–389, 1999.

[52] Daniël Knippers. Agile software sevelopment and maintainability. In *15th Twente Student Conf*, 2011.

[53] Wei Li and Sallie Henry. Object-oriented metrics that predict maintainability. *Journal of systems and software*, 23(2):111–122, 1993.

[54] Mike Loukides. *What is DevOps?* " O'Reilly Media, Inc.", 2012.

[55] Nathan Mantel. Chi-square tests with one degree of freedom; extensions of the mantel-haenszel procedure. *Journal of the American Statistical Association*, 58(303):690–700, 1963.

[56] William F Massy. Principal components regression in exploratory statistical research. *Journal of the American Statistical Association*, 60(309):234–256, 1965.

[57] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[58] David W McDonald and Mark S Ackerman. Expertise recommender: a flexible recommendation system and architecture. In *Proceedings of the 2000 ACM conference on Computer supported cooperative work*, pages 231–240. ACM, 2000.

[59] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 192–201. ACM, 2014.

[60] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, 21(5):2146–2189, 2016.

[61] Kjetil Molokken and Magen Jorgensen. A review of software surveys on software effort estimation. In *Empirical Software Engineering, 2003. ISESE 2003. Proceedings. 2003 International Symposium on*, pages 223–230. IEEE, 2003.

[62] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th international conference on Software engineering*, pages 181–190. ACM, 2008.

[63] Mark EJ Newman, Steven H Strogatz, and Duncan J Watts. Random graphs with arbitrary degree distributions and their applications. *Physical review E*, 64(2):026118, 2001.

[64] Ariadi Nugroho, Joost Visser, and Tobias Kuipers. An empirical model of technical debt and interest. In *Proceedings of the 2nd Workshop on Managing Technical Debt*, pages 1–8. ACM, 2011.

[65] Paul Oman and Jack Hagemeister. Metrics for assessing a software system's maintainability. In *Software Maintenance, 1992. Proceerdings., Conference on*, pages 337–344. IEEE, 1992.

[66] Abu Wahid Md Masud Parvez. Efficiency factor and risk factor based user case point test effort estimation model compatible with agile software development. In *Information Technology and Electrical Engineering (ICITEE), 2013 International Conference on*, pages 113–118. IEEE, 2013.

[67] Rashmi Popli and Naresh Chauhan. Cost and effort estimation in agile software development. In *Optimization, Reliabilty, and Information Technology (ICROIT), 2014 International Conference on*, pages 57–61. IEEE, 2014.

[68] Foyzur Rahman and Premkumar Devanbu. Ownership, experience and defects: a fine-grained study of authorship. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 491–500. ACM, 2011.

[69] Foyzur Rahman and Premkumar Devanbu. How, and why, process metrics are better. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 432–441. IEEE Press, 2013.

[70] Juan F Ramil. Continual resource estimation for evolving software. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 289–292. IEEE, 2003.

[71] Mehwish Riaz, Emilia Mendes, and Ewan Tempero. A systematic review of software maintainability prediction and metrics. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 367–377. IEEE Computer Society, 2009.

[72] James Roche. Adopting devops practices in quality assurance. *Communications of the ACM*, 56(11):38–43, 2013.

[73] David L Rumpf. Statistics for dummies, 2004.

[74] Panagiotis Sfetsos and Ioannis Stamelos. Empirical studies on quality in agile practices: A systematic literature review. In *Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the*, pages 44–53. IEEE, 2010.

[75] Haihao Shen, Sai Zhang, and Jianjun Zhao. An empirical study of maintainability in aspect-oriented system evolution using coupling metrics. In *Theoretical Aspects of Software Engineering, 2008. TASE'08. 2nd IFIP/IEEE International Symposium on*, pages 233–236. IEEE, 2008.

[76] Dag IK Sjøberg, Bente Anda, and Audris Mockus. Questioning software maintenance metrics: a comparative case study. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 107–110. ACM, 2012.

[77] Dag IK Sjøberg, Aiko Yamashita, Bente CD Anda, Audris Mockus, and Tore Dybå. Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering*, 39(8):1144–1156, 2013.

[78] Zéphyrin Soh, Foutse Khomh, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. Towards understanding how developers spend their effort during maintenance activities. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 152–161. IEEE, 2013.

[79] Erik Stensrud, Tron Foss, Barbara Kitchenham, and Ingunn Myrtveit. An empirical validation of the relationship between the magnitude of relative error and project size. In *Software Metrics, 2002. Proceedings. Eighth IEEE Symposium on*, pages 3–12. IEEE, 2002.

[80] Carolin Strobl, Anne-Laure Boulesteix, Thomas Kneib, Thomas Augustin, and Achim Zeileis. Conditional variable importance for random forests. *BMC bioinformatics*, 9(1):307, 2008.

[81] Muhammad Usman, Emilia Mendes, Francila Weidt, and Ricardo Britto. Effort estimation in agile software development: a systematic literature review. In *Proceedings of the 10th International Conference on Predictive Models in Software Engineering*, pages 82–91. ACM, 2014.

[82] Huanjing Wang, Taghi M Khoshgoftaar, and Naeem Seliya. How many software metrics should be selected for defect prediction? In *FLAIRS Conference*, 2011.

[83] Haiyun Xu, Jeroen Heijmans, and Joost Visser. A practical model for rating software security. In *Software Security and Reliability-Companion (SERE-C), 2013 IEEE 7th International Conference on*, pages 231–232. IEEE, 2013.

[84] Yuming Zhou and Baowen Xu. Predicting the maintainability of open source software using design metrics. *Wuhan University Journal of Natural Sciences*, 13(1):14–20, 2008.

[85] Donald W Zimmerman and Richard H Williams. The relative error magnitude in three measures of change. *Psychometrika*, 47(2):141–147, 1982.

Appendix A

Code Quality Metrics from the Tool Understand

This appendix contains a list of the code quality metrics.

Complexity Metrics: Average Cyclomatic Complexity, Average Modified Cyclomatic Complexity, Average Strict Cyclomatic Complexity, Average Essential Cyclomatic Complexity, Average Essential Strict Modified Complexity, Number of Paths, Cyclomatic Complexity, Modified Cyclomatic Complexity, Strict Cyclomatic Complexity, Essential Complexity, Essential Strict Modified Complexity, Knots, Max Cyclomatic Complexity, Max Modified Cyclomatic Complexity, Max Strict Cyclomatic Complexity, Max Essential Complexity, Max Knots, Max Essential Strict Modified Complexity, Depth of Inheritance Tree, Nesting, Minimum Knots, Comment to Code Ratio, Sum Cyclomatic Complexity, Sum Modified Cyclomatic Complexity, Sum Strict Cyclomatic Complexity, Sum Essential Complexity, Sum Essential Strict Modified Complexity

Volume Metrics: Average Number of Blank Lines, Average Number of Lines of Code, Average Number of Lines with Comments, Blank Lines of Code, Lines of Code, Lines with Comments, Average Number of Lines, Number of Function, Internal Instance Variables, Protected Internal Instance Variables, Friend Methods, Local Internal Methods, Local Protected Internal Methods, Inputs, Physical Lines, Blank Lines of Code, Semicolons, Statements, Comment to Code Ratio

Object Oriented Metrics: Number of classes, Coupling, Number of Children, Class Methods, Class Variables, Instance Methods, Instance Variables, Private Instance Variables, Protected Instance Variables, Public Instance Variables, Local Methods, Methods, Local Const Methods, Local Default Visibility Methods, Friend Methods, Private Methods, Protected Methods, Public Methods, Local strict private methods, Local strict published methods, Modules, Program Units, Subprograms, Depth of Inheritance Tree, Lack of Cohesion in Methods

Appendix B

Tracer: Implementation and Data Visualization

While the architecture itself (2.5) is simple, implementation and optimization were not as straightforward. Before the client can be used, authentication through both VSTS (OAuth 2.0) and the client itself are a necessities. The Shell client only interacts with the Tracer server using a rest API. The load on the client side is very light and all calculations are primarily done on the server. The client uses morris.js[1] to visualize the charts. The back-end consists of ExpressJS [2] (a node.js framework), Mongoose (to interact with the database) and small pieces of code that are hard to realize in Javascript are written in R.

As previously mentioned, the structure and data of VSTS was not suitable for analytics (some charts would take hours to calculate), hence we choose to create a back-end (tracer server). The amount of data, however, is rather large (the database currently is sized close to 25GB), and fetching frequent queries in real time with limited computational power can be difficult, primarily because all the data also have to contain the history of the projects. We will show some examples of difficult to optimize charts in the next section. Consider a more detailed image of the architecture of the 'Tracer Server' from Figure 2.5 in Figure B.1.

In essence, the calls to the REST API consist of database queries and the results to the query are simply returned to the user. Lists of queries are also accepted. This results in a limited amount of chatter between the front and the server. In some scenarios optimization is required, by which the API calls consist of only a couple of parameters and the server will then return an optimized response.

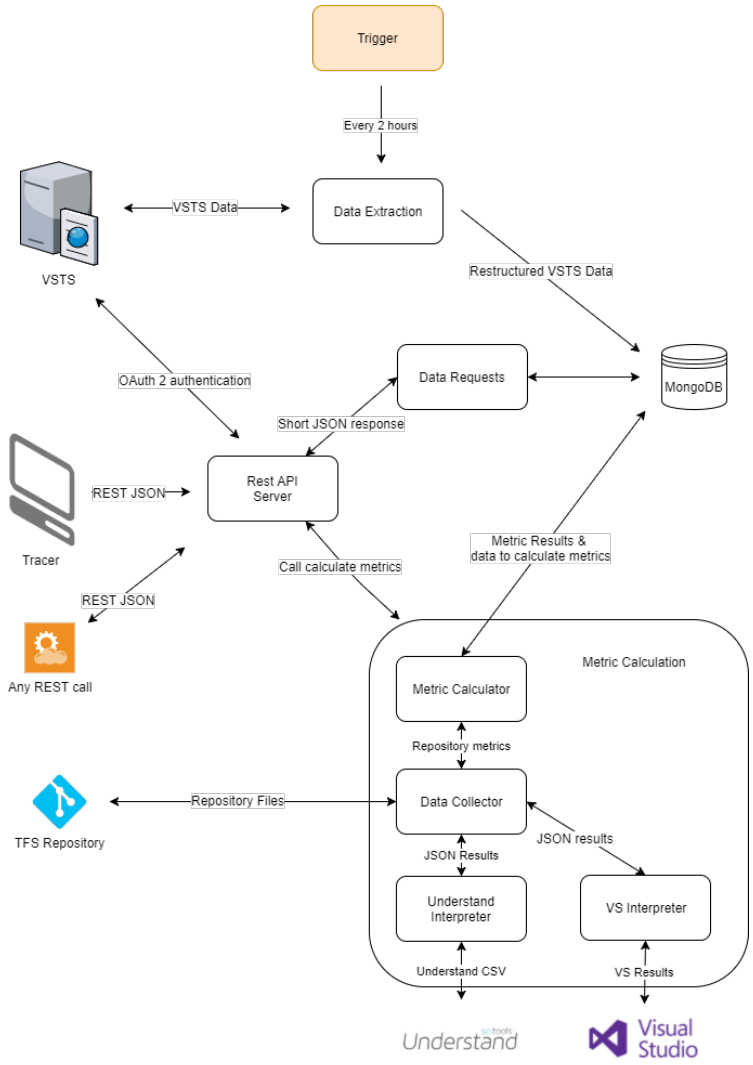
Because we store nearly all data that are available in VSTS, we have the option to filter out any form of activity performed by a single user to determine if he or she was active on VSTS at the given time. This information is the source of the user activity in Section 2.9.

To enhance security, the features are only available within the Shell intranet. Because the platform mostly serves to review existing data, the synchronization between VSTS and the server is not instantaneous. Every two hours the server pulls the latest changes from

[1] `http://morrisjs.github.io/morris.js/`

[2] `https://expressjs.com/`

Figure B.1: Tracer Server Architecture



VSTS (the Trigger in Figure B.1) and adds them to the database. The data are restructured to serve the analytic purposes of Tracer and stored in a MongoDB database.

The metrics are calculated in two ways. The first just pulls the complete repository, calculates the metrics and exports them. Then it pulls the difference and repeats. Another (quicker) approach pulls the complete file history for a single file, calculates the metrics, exports them and repeats the step. Some process and change metrics require information that is already part of the database. Therefore, the database is also accessed to calculate, for example, the number of versions of the file. All the results are stored in a temporary database that is then exported as CSV. This CSV file will serve as input for the calculations in R (Section 2.10).

We split the visualization of the metrics into two categories; defects and backlog. The

defect page is primarily involved in the representation of defects. Backlog refers to the agile backlog and is more involved in agile metrics. There is a general overview page that compares all the metrics from the projects in a heatmap, but we won't discuss it in this section. These metrics are primarily a product of user (within Shell) input or obtained from existing literature. These charts were also used during the project selection procedure to identify for example consistency (section 2.3). There are twenty graphs in total and about fifteen numerical metrics. In this section, we present some interesting graphs.

B.1 Backlog

The backlog metrics revolve around the visualization of the agile backlog behavior. The metrics can be filtered based on projects, team and sprint. Sprint is not a necessity because filtering can also be done between two dates. In this section we provide a brief overview of some of the charts.

Velocity

Velocity is the measure of the amount of work that can be tackled or has been tackled during a time period. These charts will frequently be calculated for a sprint. We have two separate interpretations for velocity. Often velocity will be interpreted by calculating the sum of the story points assigned to a PBI. We also consider an implementation in which the velocity is the sum of the hours assigned to a task. This gives insight into the estimated time into story points for a sprint and the actual hours spent. We choose to represent these values in a bar charts, as shown in Figure 4.2.

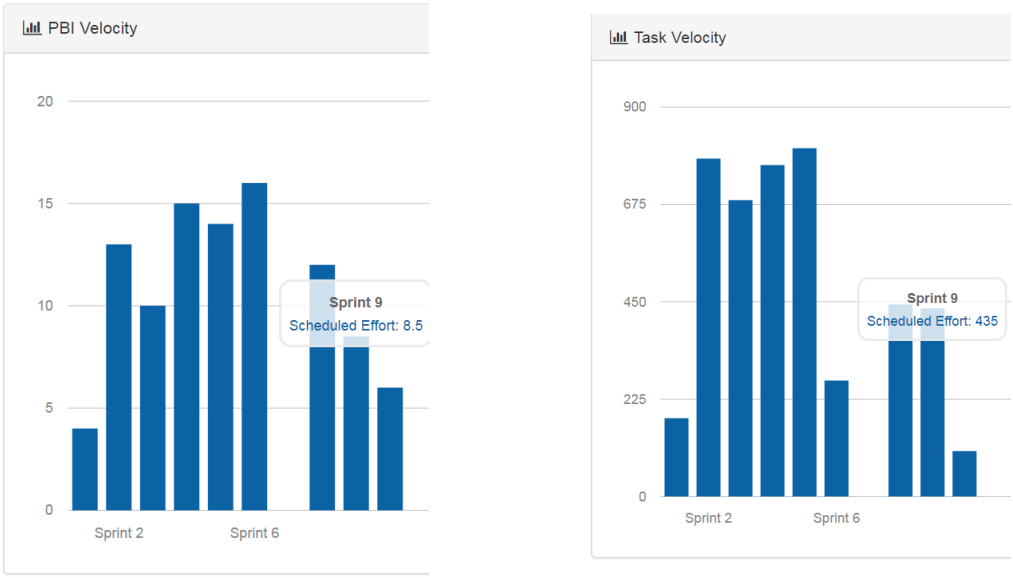
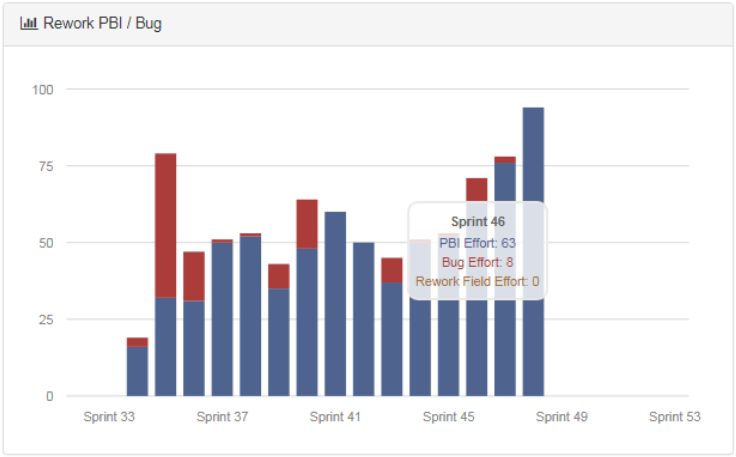


Figure B.2: Left: PBI Velocity Right: Task velocity for the first 10 sprints

These two variations open up possibilities to review the actual hours spent on a PBI or hours spent for every individual story point (sum of tasks hours for PBI/story points for PBI). This could ease the identification of PBIs, which took a lot longer than estimated. We present these metrics in a table for every individual PBI. These velocity charts do not directly incorporate the story points that were required to complete a bug. Therefore, we create a rework chart. A chart that contains the sum of the story points of the PBIs, the bugs and an optional 'rework field'. The rework field can be part of a PBI and is the amount of rework required (in story points) to complete the PBI. Figure B.3 shows an example of a rework chart.

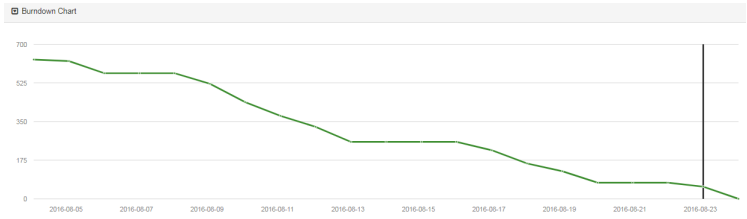
Figure B.3: Rework Chart



Burn and flow charts

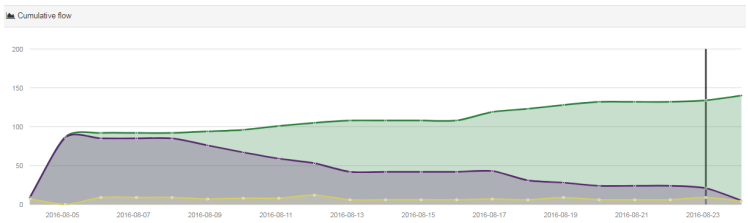
To visualize the progression during a time period, frequently a sprint, one can use a burn-down, burn-up or a cumulative flow chart. These charts represent the flow of time or the statuses of the tasks/PBIs at a certain moment in time. An example of a burndown chart for a task can be observed in Figure B.4 and the cumulative flow for tasks in Figure B.5.

Figure B.4: Task burndown



The same charts can be generated for PBIs or any other work item (bug, impediments). These can help identify moments of setback in the project. They also make it possible to identify pitfalls in the process, if for example a developer chooses to perform all his or her

Figure B.5: Task cumulative flow



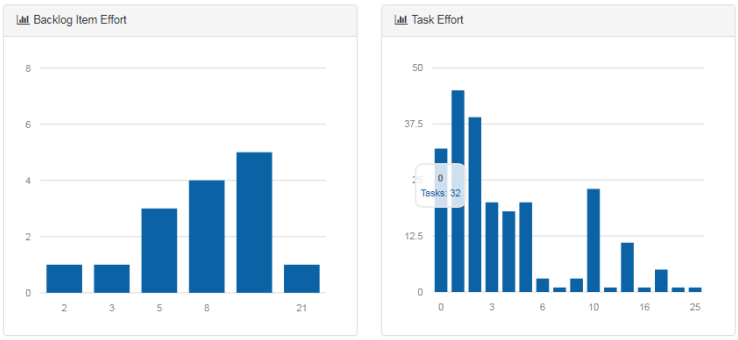
task maintenance at one and the same point in time. The initial plan was to also implement lead and cycle time, but Microsoft already deployed very neat versions of this during the development of the platform.

This can be a good time to reflect on the complexity of the calculations. The examples in Figures B.4 and B.5 only contain 140 tasks. This number can increase very quickly if one chooses not to filter on sprints (like this example) but between two dates (e.g., a year). Every work item will have a number of revisions in which some sort of change was introduced. A task, for example, has 30 revisions on average and a PBI of about 150. The user can choose how many intervals he or she wants to view on the chart (e.g., 20 in the above examples). We have to optimize the application in such a way that in real time, the application is able to retrieve the state and effort in these work items for multiple points in time. While in essence it is not a complicated problem, certain difficulties arise when the data-set is large and the computational power is limited.

Distributions and effort

If we wish to identify cases in which developers neglect the effort field, because they for example just fill in 0 everywhere, a distribution is a good method. The same is applicable to PBIs and story points. Consider Figure B.6, which represents the distribution of the task and PBI effort.

Figure B.6: PBI and Task distributions



More charts can be realized for effort. People estimate a certain number of story points to a PBI, but it is difficult to trace how much time was actually spent on development. We

sum up the effort of all the tasks in a PBI (the same as our methodology in section 2.7). In a table, one can than see an overview of the total effort spent on every PBI or bug, and how it compares to the total number of story points assigned to the work items. We also provide the user with information about the number of tasks it took to complete the PBI, either in the selected sprint or in all the sprints.

Other metrics

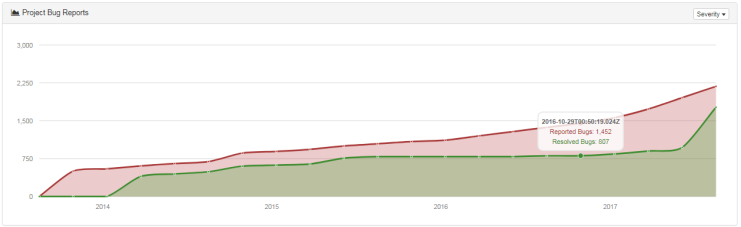
Every work item can contain multiple descriptions. A PBI for example has a description and acceptance criteria (in words). We count the words of these descriptions and using a rating [83] system we attempt to rate the textual content of the work items.

In a normal scenario it should be possible to complete a PBI in one single sprint. However, in practice these PBIs may be moved from one sprint to another. This sometimes happens a multitude of times. To identify these cases, we simply add a field to the table that provides the number of unique sprints a PBI was part of. The user could then inspect these PBIs, and validate the number of story points and actual hours spend to see if planning mistakes were made. To further identify the longevity of the PBI we present the time it took to complete the PBI from both the created and the 'in progress' date.

B.2 Defects

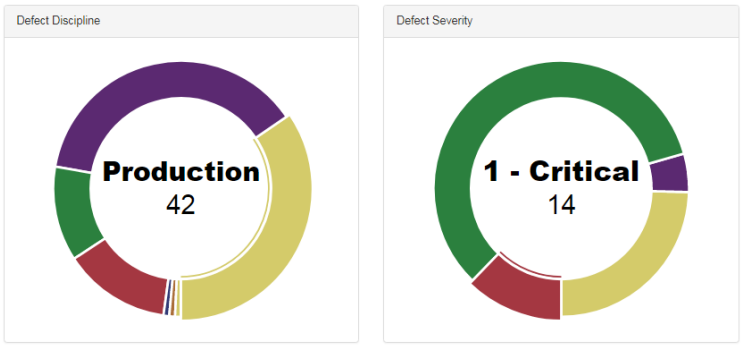
It is of course interesting to see how many bugs are reported and resolved at any point of time. Consider the chart in Figure B.7 which displays the total number of bugs reported and resolved over the past four years for a project.

Figure B.7: Reported and resolved bugs in the last four years.



We create comparable distributions and effort charts as we did for the PBI. Other than providing the word count and rating for the description and acceptance criteria we also include the words for the fix description. The charts for the task and its effort, distributions and the burn-charts are also available for the defects. In figure B.7 one can see a small drop-down to select the severity of the defects. Teams can manually define the severity of the bug and often are identified in four categories: low, medium high or critical. Teams also use a label to categorize the bug to for example let the team know it was a production, test or deployment defect. We represent these using a donut chart as can be seen in figure B.8.

Figure B.8: Reported and resolved bugs in the last four years.



Furthermore there is an overview available of the most complicated defects during the selected time period. Complicated means that they took the longest time to fix. One can see an overview of the defects for a given sprint, but also between a selected time periods.

Appendix C

Interview Questions

Please note that during all questions improvisation occurred. This is just a guideline. During a lot of questions we ask for examples.

C.1 General questions

- When you create the efforts for tasks, do you do it with your SCRUM team? Or individual?
- How do you create an estimate for a task?
- What factors influence your estimation?
  - How would you measure ...?
- Before doing a task estimation do you think about the files/methods you have to modify?
  - [YES] Only files or also methods?
  - [YES] Do you inspect these files/methods?
  - [YES] Do you think these considerations increase your accuracy?
  - [NO] Why not? If you would do you think it would increase the accuracy of your estimation?
- Do you take Code Quality and complexity (elaborate) into account when doing an estimation?
  - [YES] What kind of impact does it have on your decision? Why?
  - [NO] Why not? Do you think code quality has a negative impact on maintainability?
  - Do you feel like bad code quality has an impact on the speed at which you develop? How severe and why?

        * [YES] Can you give an example where bad code quality has an impact on task development speed?

- Do you take Design Smells (elaborate) into account when doing an estimation?
  - [YES] Does it have a big impact on your decision? Why?
  - [YES] Is the impact greater than that of Code Quality? Why?
  - [NO] Why not, don't you think this has a big impact?
  - Do you feel like design smells (could) have an impact on the speed at which you develop? How severe and why?
  - Do you use SonarQube?
    * How do you use it?
    * Do you consider SonarQube when making estimates? Why (not)?
    * Would you say the improved code quality, as a result of SonarQube, allows you to deploy new features faster? Why?
    * How many time do you spend don keeping SonarQube results up to standard?
- Do you take the history and your familiarity (elaborate, process metrics) with the file into account when doing an estimation?
  - [YES] Does it have a big impact on your decision? Why?
  - [YES] Is the impact greater than that of Design Smells?
    * [NO] What about code quality?
  - [NO] Don't you think your experience with the specific files would influence your development speed?
  - [NO] What about the history of the file?
- What other factors do you consider when creating an estimation?
  - Do you consider these factors to have a bigger impact than the code quality, process metrics or design smells? Why?
  - Do you consider the total size of the change? How would you measure the size of a change?
- Do you perceive a difference between your estimate and your actual development time? Can you give an example of where you went over the limit of the hours?
- Where do you frequently encounter quality issues?
  - How do you deal with them?
  - What do you do if you did not estimate the time for it?
- How do quality issues influence your effort estimations?
- Do you ever feel like your estimations suffer because of code quality?

C.2 Tasks

High effort - bad quality:

- What was your reasoning before you estimated the x hours for this task?
  - Were you familiar with the code?
  - Did this familiarity impact your estimation? Why?
- Do you think the quality of the modified code had an effect on your estimation?
  - [*YES*] What quality factors did you take into account? Ask about CQ, Process, Design
  - [*NO*] Do you think your estimate would have been more accurate if you did? Why (not)? Why not do it?
- Do you think the quality of the code had an effect on your actual time spend on the task? Why?
  - [*YES*] Ask to put in order of severity: Code Quality, Design Smells, Process Metrics
- What other factors did you take into account when estimating the x hours for this task?
- Do you think your estimate was accurate?
  - [*BAD*] What made the time spend deviate from your estimation?

Intermediate estimation

Questions:

- What was your reasoning before you estimated the x hours for this task?
  - Were you familiar with the code?
  - Did this familiarity impact your estimation? Why?
- Why did you create an intermediate estimation?
- Do you think your estimation would be more accurate because your familiarity increased?
- Is this intermediate estimations more impacted by the quality of the code?
  - Ask what had the highest impact; process, code quality, design smells. Why?
  - What about intermediate estimates in general?
- What other factors did you take into account when estimating the x hours for this task?

Revised estimation

- Why did you revise your estimation?
- [If not CQ] Do you ever revise your estimation because you realize the quality of the code is worse than expected?
- Do you think your revised estimates are more accurate?
  - [YES] Is it because you keep the quality into account?
  - Why/what else influenced it?
- Do you consider it important to have an accurate estimation before you start a task?

Appendix D

Additional Regression Results

D.1 PBI task effort and code Quality Metrics

Table D.1: Technical Code Quality and Process Metrics

	Technical Code Quality Metrics				Process Metrics			
Project	PCR	ρ	SEL	ρ	PCR	ρ	SEL	ρ
Project 1	0.11	0.00	0.10	0.00	0.48	0.00	0.39	0.00
Project 2	0.05	0.10	0.07	0.16	0.23	0.00	0.19	0.00
Project 3	0.12	0.01	0.18	0.02	0.32	0.03	0.34	0.00
Project 4 Team 1	0.1	0.02	0.23	0.03	0.40	0.00	0.33	0.02
Project 4 Team 2	0.01	0.36	0.05	0.40	0.49	0.00	0.35	0.00
Project 4 Team 3	0.04	0.22	0.02	0.37	0.15	0.45	0.07	0.42

Table D.2: Combined Metrics

	Combined Metrics			
Project	ρ	SEL	ρ	
Project 1	0.4	0.01	0.39	0.04
Project 2	0.23	0.01	0.35	0.00
Project 3	0.29	0.04	0.42	0.02
Project 4 Team 1	0.39	0.02	0.41	0.00
Project 4 Team 2	0.32	0.00	0.42	0.00
Project 4 Team 3	0.18	0.09	0.18	0.06

D.2 Story points and Code Quality Metrics

Table D.3: Technical Code Quality and Process Metrics

	Technical Code Quality Metrics				Process Metrics			
Project	PCR	ρ	SEL	ρ	PCR	ρ	SEL	ρ
Project 1	0.00	0.85	0.03	0.59	0.35	0.00	0.31	0.02
Project 2	0.11	0.12	0.15	0.15	0.17	0.01	0.25	0.07
Project 3	0.08	0.35	0.02	0.29	0.25	0.18	0.19	0.23
Project 4 Team 1	0.1	0.45	0.28	0.12	0.32	0.00	0.23	0.04
Project 4 Team 2	0.02	0.23	0.25	0.41	0.31	0.00	0.34	0.00
Project 4 Team 3	0.05	0.36	0.01	0.29	0.09	0.45	0.01	0.35

Table D.4: Combined Metrics

	Combined Metrics			
Project	ρ	SEL	ρ	
Project 1	0.07	0.08	0.30	0.70
Project 2	0.11	0.15	0.09	0.12
Project 3	0.17	0.00	0.33	0.01
Project 4 Team 1	0.19	0.04	0.32	0.00
Project 4 Team 2	0.29	0.00	0.37	0.00
Project 4 Team 3	0.14	0.40	0.15	0.52