Assessing the viability of implicitly estimated velocity for measuring the productivity of software teams

Max Pagels

MSc thesis
UNIVERSITY OF HELSINKI
Department of Computer Science

Helsinki, May 3, 2013

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author
Max Pagels

Työn nimi — Arbetets titel — Title

Assessing the viability of implicitly estimated velocity for measuring the productivity of software teams

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
MSc thesis	May 3, 2013	65

Tiivistelmä — Referat — Abstract

Productivity is an important aspect of any software development project as it has direct implications on both the cost of software and the time taken to produce it. Though software development as a field has evolved significantly during the last few decades in terms of development processes, best practices and the emphasis thereon, the way in which the productivity of software developers is measured has remained comparatively stagnant. Some established metrics focus on a sole activity, such as programming, which paints an incomplete picture of productivity given the multitude of different activities that a software project consists of. Others are more process-oriented — purporting to measure all types of development activities — but require the use of estimation, a technique that is both time-consuming and prone to inaccuracy. A metric that is comprehensive, accurate and suitable in today's development landscape is needed.

In this thesis, we examine productivity measurement in software engineering from both theoretical and pragmatic perspectives in order to determine if a proposed metric, *implicitly estimated velocity*, could be a viable alternative for productivity measurement in Agile and Lean software teams. First, the theory behind measurement — terminology, data types and levels of measurement — is presented. The definition of the term productivity is then examined from a software engineering perspective. Based on this definition and the IEEE standard for validating software quality metrics, a set of criteria for validating productivity metrics is proposed. The motivations for measuring productivity and the factors that may impact it are then discussed and the benefits and drawbacks of established metrics — chief amongst which is productivity based on lines of code written — explored.

To assess the accuracy and overall viability of implicitly estimated velocity, a case study comparing the metric to LoC-based productivity measurement was carried out at the University of Helsinki's Software Factory. Two development projects were studied, both adopting Agile and Lean methodologies. Following a linear-analytical approach, quantitative data from both project artefacts and developer surveys indicated that implicitly estimated velocity is a metric more valid than LoC-based measurement in situations where the overall productivity of an individual or team is of more importance than programming productivity. In addition, implicitly estimated velocity was found to be more consistent and predictable than LoC-based measurement in most configurations, lending credence to the theory that implicitly estimated velocity can indeed replace LoC-based measurement in Agile and Lean software development environments.

ACM Computing Classification System (CCS):

D.2.9 [Software Engineering]: Management—Productivity,
D.2.8 [Software Engineering]: Metrics

Avainsanat — Nyckelord — Keywords
software productivity, software metrics, productivity measurement

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Contents

1 Introduction

Productivity is an important factor when considering the cost of developing software [Boe87]. Striving to produce software in shorter periods of time results in fewer person-months spent, which in turn reduces financial cost.

Improving productivity can be seen to consist of three main strategies: automating what can be automated, to speed up development; avoiding work of little value, to work in a smarter fashion; and reusing software components to reduce unnecessary work [Boe99]. In order to determine if productivity is at an acceptable level or even improved compared to prior efforts, it must be measured.

Productivity is typically defined as the ratio of what is produced (*output*) to the amount of time required to produce it (*effort* or *input*). In the field of software development, measuring productivity has proven to be a challenge, mainly due to the individual nature of any given software project and the abstract nature of software itself. Established productivity metrics reflect this challenge: some metrics are only intended to measure one specific activity of a software development project (such as programming), whereas others are capable measuring productivity across all conceivable activities but require the use of time-consuming estimation.

Ideally, the productivity of a software team or its members should be measured using a metric that is flexible enough to take all possible nuances and types of activities in a software project into account whilst still being at least as accurate as the metrics established to date. The data needed to assess productivity should be simple to collect yet minimally obtrusive so that it does not impede the day-to-day work of a software development team.

Software engineering as a field has seen significant changes since its origins. Since the first international conference on the subject in 1968 [Mah04], processes, models and best practices have evolved considerably. In contemporary software engineering, it is universally accepted that high quality software

routinely results from high quality development processes [MAKS12]. As such, a productivity metric should not only fulfill the aforementioned requirements, but also be suitable for use in the development processes adopted today.

Given the proliferation of Agile and Lean methodologies, it is relevant to examine not only how established productivity metrics fare in such a environment, but also if it is possible to develop a new metric that suitable for use in such methodologies. In this thesis, we propose a new metric — *implicitly estimated velocity* — and compare its usefulness and accuracy to that of arguably the most well-known productivity metric, productivity based on lines of code. In particular, we discuss the following three questions:

1. Is productivity based on lines of code a valid metric for productivity in Agile and Lean software development?
2. Is the proposed concept of implicitly estimated velocity a valid metric for productivity in Agile and Lean software development?
3. If productivity based on lines of code is unsuitable in an Agile and Lean environment, could implicitly estimated velocity serve as a viable replacement?

In order to address the aforementioned questions, the results of a case study of two software projects have been analysed and interpreted. The projects were carried out at The University of Helsinki's Software Factory [Sof13], a workplace-like environment in which small teams of students create software applications during a span of approximately seven weeks. The Factory allows researchers to perform experiments and collect data to study almost any aspect of a software project.

This remainder of this work is organised as follows. Chapters 2 and 3 provide information on measurement theory and productivity measurement in software engineering, respectively. Chapter 4 describes the motivations and practicalities of measuring productivity. Chapter 5 presents established

metrics, whilst Chapter 6 defines the proposed metric of implicitly estimated velocity. Chapters 7 to 9 describe the design and execution of the Software Factory case study. The study results are presented in Chapter 10. Chapter 11 summarises the work with conclusions and discussion.

2 Measurement theory

In order to successfully measure an entity and correctly interpret the results of a measurement, knowledge of measurement theory, terminology and data types are prerequisite. Measurement theory can be broken down into two related concepts: attributes and metrics.

2.1 Attributes

An *attribute* can be defined as "a measurable physical or abstract property of an entity" [iee98]. Intuitively, an attribute is a property of a software product, such as reliability. An attribute may also describe some property of a software development process, such as team member productivity. To successfully measure an attribute, one must derive a suitable metric for it.

2.2 Metrics

A *metric* — occasionally referred to as a *measure* — describes what information should be collected to measure a particular attribute, and how this information can be calculated to yield a meaningful result. For example, a metric for software size could simply be the total number of source lines of code (LoC). Such a metric tells us that to determine the size of an application, we must obtain the application source code and total the number of code lines it comprises.

Metrics can be used for both *direct measurement* and *indirect measurement*. A direct metric is a metric that is only dependent on a single variable; LoC as a metric of size is an example of a direct metric. An indirect metric is dependent on two or more variables. Indirect metrics are commonly used to measure more complex attributes such as productivity, whose generic definition is the amount of input or effort required to produce a unit of output — the ratio of two variables.

2.3 Data types

After recognising a set of attributes for measurement and deriving a corresponding set of valid metrics, the data subject to measurement must be collected. The applicable methods for data collection vary depending on the data type (qualitative or quantitative) as well as its level of measurement (nominal, ordinal, interval or ratio).

Qualitative data is non-numerical data that can be categorised using a nominal scale. Using a nominal scale, the only permitted empirical operation is the determination of equality [Ste46]. Typical examples of such data include binary data (yes or no, good or bad) and categorical data (belonging to one particular category). Qualitative data is generally collected using methods such as surveys and interviews.

Quantitative data is numerical data that follows an ordinal, interval or ratio scale. Compared to a nominal scale, an ordinal scale allows for one additional operation: the determination of greater or less [Ste46]. Data of this form can thus be ordered. A example of an ordinal scale is the Mohs scale of mineral hardness [MSBD04].

Using an interval scale, the following operations are permitted:

- Determination of equality.
- Determination of greater or less.

- Determination of equality of intervals or differences.

Intuitively, data following an interval scale is data that lacks a true *zero point*. A common example of such a scale is the scale of temperature in degrees Centigrade, in which the zero point is a point that has been agreed upon — zero degrees Centigrade does not imply "no temperature".

A ratio scale is the most powerful in terms of applicable statistical methods. Using a ratio scale, the following operations are permitted [Ste46]:

- Determination of equality.
- Determination of greater or less.
- Determination of equality of intervals or differences.
- Determination of equality of ratios.

One example of quantitative data using a ratio scale is LoC. Though quantitative data can be collected using the same methods described above, it is often possible to automate the entire collection process. For the case study presented in this thesis, only quantitative data is analysed; the level of measurement is mentioned beforehand for each of the quantitative data sets used.

Qualitative and quantitative *data* are not to be confused with qualitative and quantitative *research*. Qualitative research is a research method that focuses less on numbers and more on the human elements of a topic [KM86]; quantitative research is more focused on numbers analysed using mathematical, statistical and other computational techniques [Giv08].

3 Productivity measurement in software engineering

In this thesis, *productivity* is defined as a measurable attribute of a software development team or its members that describes the total output produced during a given span of time. As such, its core focus not on software itself, but rather the people who produce it. As with any other metric, a productivity metric should not only be well-defined, but also validated before use.

3.1 Definition

Figure 1 presents a generic linear function for productivity — the ratio between the output o of a software team (or software team member) and their effort i.

$$f(o, i) = \frac{o}{i}$$

Figure 1: A generic productivity function.

The aforementioned function is by far the most widely used model for productivity in software engineering [BSVW96, BEEB98, JRW01, BBM96], and the basis upon which the most common productivity metrics have been developed.

3.2 Validating productivity metrics

In order to ensure that a given metric measures what it is intended to measure, it must be validated. A metric can be seen as a mathematical function; as such, any metric can be validated mathematically. In software engineering, a metric can be defined as a function whose inputs is software or software process data and whose output is a single numeric value [iee98].

For attributes pertaining to software quality, a direct metric can be considered valid if and only if it satisfies the following criteria [KMB04, iee98]. Here, *attribute values* are values calculated from a different metric acknowledged to indicate software quality.

1. *Correlation*: a metric M should be linearly related to the attribute it is supposed to measure, as measured by the statistical correlation between the metric values and corresponding attribute values.

2. *Consistency*: if values of the attribute A under study are monotonic (i.e., $A_1 > A_2 > ... > A_n$), then the corresponding metric values $M_1, M_2, ..., M_n$ should also be monotonic ($M_1 > M_2 > ... > M_n$). Intuitively, this criterion describes if the metric can accurately rank a set of products or processes by quality.

3. *Tracking*: if a metric M is related to the attribute A it it supposed to measure, then if the attribute value changes from A_1 to A_2, the metric value should also change from M_1 to M_2 in the same direction (i.e., if $A_1 > A_2$, then $M_1 > M_2$). As such, this criterion describes to which extent the metric in question is capable of tracking changes in a product's quality at different points in time.

4. *Predictability*: if the result of the metric's function F is known at any given point in time, one should be able to predict the result of F for any given time in the future.

5. *Discriminative power*: a metric should clearly discriminate between good results and poor results. For example, if a metric measures software reliability, a highly reliable application should yield a result significantly higher than an application with low reliability.

6. *Reliability*: a metric should satisfy criteria 1-5 for at least N per cent of the times it is applied. Intuitively, this means that the higher the percent of times all validation criteria are satisfied when applying the metric, the higher the reliability of the metric itself.

The criteria presented above are designed for direct software quality metrics. They can, however, be adapted to assess productivity, an indirect construct. Using the generic productivity function $f(o, i) = \frac{o}{i}$ as a base, we propose the following validity criteria:

(i) *Correlation*: a productivity metric M should be linearly related to the attribute it is supposed to measure, as measured by the statistical correlation between the metric values and corresponding attribute values.

(ii) *Consistency*: given a sequence P of attribute values $P_1, P_2, ..., P_n$ for a software project, the corresponding metric values $M_1, M_2, ..., M_n$ should be monotonic — $M_1 > M_2 > ... > M_n$ — if $P_1 > P_2 > ... > P_n$. Intuitively, this criterion describes if a productivity metric can accurately rank the productivity of members of a software project.

(iii) *Tracking*: if a productivity metric M is indeed related to the attribute of productivity P, then if productivity changes from P_1 to P_2, the metric value should also change from M_1 to M_2 in the same direction (i.e., if $P_1 > P_2$, then $M_1 > M_2$). As such, this criterion describes to which extent the productivity metric in question is capable of tracking changes in productivity at different points in time.

(iv) *Predictability*: if the result of the productivity metric's function F is known at any given point in time, one should be able to predict the result of F for any given time in the future.

(v) *Discriminative power*: a productivity metric should clearly discriminate between good results and poor results.

(vi) *Reliability*: a productivity metric should satisfy criteria i)-v) for at least N per cent of the times it is applied. Intuitively, this means that the higher the percent of times all validation criteria are satisfied when applying the metric, the higher the reliability of the metric itself.

For the aforementioned criteria, results from a generally accepted productivity metric other than the one under inspection can be used as attribute values. This, however, is not a requirement: if the validity of established metrics themselves are under inspection, one may use an alternative data source such as expert opinion or assessment by one of more of a software project's stakeholders. The case study in this work uses peer-assessment by project team members to constitute such a data source (see Chapter 7.3).

As with the criteria for direct software metrics, we require that all of the aforementioned criteria are satisfied for a productivity metric to be considered valid. The following mathematical and statistical methods can be used to determine if a criterion is satisfied:

(i) *Correlation*: to measure the linear dependence between two variables of a different scale, the Pearson product-moment correlation coefficient can be used.

(ii) *Consistency*: in order to determine the degree to which a metric is consistent, one can calculate Spearman's rank correlation between attribute values and their corresponding metric values.

(iii) *Tracking*: in order to determine the degree to which a metric fulfills the tracking criterion, one can calculate Spearman's rank correlation over n attribute and metric value pairs (P_i, M_i), where P_i and M_i have been measured at the same point in time.

(iv) *Predictability*: standard deviation can be used to determine to which extent values vary in a mathematical sequence. The lower the variation, the easier it becomes to predict the value v_n in a series given the preceding values $v_0, ..., v_{n-1}$. To be able to compare the deviations of multiple series with differing units, one must first normalise the deviation results by diving the mean of a series by its standard deviation. Such a normalised standard deviation of often referred to as a *coefficient of*

variation [Bro98]. The lower the unbiased[1] coefficient of variation, the higher the predictability of the metric under evaluation.

(v) *Discriminative power*: the discriminative power of a metric can be assessed by calculating the unbiased coefficient of variation of a mathematical sequence. Unlike when assessing predictability, the higher the value of the coefficient, the greater the discriminative power.

(vi) *Reliability*: to assess the reliability of a metric, one may simply divide the times the metric satisfies criteria i)-vi) by the total number of times the metric is applied.

The methods for checking the validity criteria produce a numeric result. This requires determining suitable *threshold values* which much be either achieved or exceeded for the criterion in question to be satisfied. This allows for a degree of customisation: in some contexts, the discriminative power of a metric may be of more importance than its predictability, for instance. In such cases, threshold values can adjusted accordingly.

4 Productivity measurement in practice

To successfully assess the productivity of a software team or its members, one must not only check the validity of the applied metric, but also understand why and for what purpose such measurements can be carried out. In addition, one must be able to correctly interpret the results, which entails understanding what can impact productivity and to which extent.

[1] Coefficients of variation are typically computed on samples that estimate an underlying population. Such biased coefficients can be converted into unbiased coefficients using the formula $(1 + \frac{1}{4N})C$, where N is the sample size and C the biased coefficient [Bro98].

4.1 Reasons for measuring productivity

Productivity data is collected and analysed for a variety of different reasons; these reasons typically differ depending on who functions as the stakeholder, i.e. from whose viewpoint productivity is being examined. Collecting productivity data pertaining to software engineering is beneficial to any given organisation, and can be motivated with a basic example.

Table 1 presents some possible benefits of productivity measurement in a scenario where data related to the productivity of a development team and its members has been collected and analysed against to a given metric. Assuming a valid metric, the results are meaningful to different stakeholders in different ways.

Stakeholder	**Possible benefit(s)**
Development team member	personal benchmark; validation of individual effort
Development team lead	leadership benchmark; validation of managerial skills; baseline for improvement
Software development lead	comparison point; leadership benchmark; validation of managerial skills; baseline for improvement
Business lead	comparison point; cost analysis; comparison point; baseline for improvement

Table 1: Possible benefits of team productivity measurement.

For a development team member, his or her personal productivity data could be used as a benchmark to validate current effort, or seen as an encouragement to heighten productivity.

For the team lead, the the combined productivity of the team could be used to assess his or her managerial and leadership skills. The individual productivity of team members could be analysed to improve the combined productivity in future projects, for example by introducing a different software development

process or reassigning specific development work to different persons.

For the software development lead, productivity data for different teams could be compared and used for improvement: intensive projects can be assigned to proven highly productive teams and software projects in a particular domain to teams with a proven track record in that domain.

For the business lead, the main motivator for measuring software engineering productivity is that it has a direct impact on cost: a highly productive team can produce results faster, reducing overall costs in terms of man hours needed to complete a project.

In a software development project, there may exist several stakeholders other than those mentioned above. For the results of productivity measurement to be interpreted correctly, knowledge of the factors that impact productivity is needed regardless.

4.2 Factors that impact productivity

Given a productivity metric, the interpretation of its results for the purpose of cross-team or cross-context comparison requires comprehensive knowledge of the impact factors present in that scenario. When discussing productivity metrics, an impact factor is a factor that in some form impacts how productive development team members are when working on a project. Impact factors contribute either positively or negatively to productivity ratings, and often vary between projects of different nature.

The sheer amount of different impact factors in software engineering necessitates some form of abstraction or simplification. Vosburgh et al. suggest separating productivity factors into *project-related factors* and *product-related factors* [VCW+84]. Project-related factors describe how an application is developed; such factors can be controlled by management personnel. Product-related factors, on the other hand, cannot be controlled by management

staff. As such, product-related factors can be regarded as constant over project-related variables [VCW+84]. Intuitively, this means that given a set of product-related factors, the set would not change even if the values of project-related variables were to change. Table 2 presents some key project- and product-related impact factors.

Trendowicz et al. present a different abstraction where factors influencing productivity are separated into *context factors* and *influence factors* [TM09]. A context factor is a factor pertaining to the context or environment of a given software engineering project, whereas an influence factor is a factor impacting productivity within a given context.

Table 3 presents some notable impact factors found during a study of 126 publications, 13 industrial projects, eight surveys and four workshops [TM09]. In most cases, it is clear that influence factors are determined by context factors: for a given project, its significant impact factors may vary depending on the overall context.

4.3 Comparing productivity across projects

Due to the vast amount of factors that can impact productivity, comparing measurement results across projects may be highly misleading if done without properly adjusting the results to take such factors into account. Given two projects, one of two scenarios may occur [TM09]:

- The set of impact factors for each project may overlap significantly. Intuitively, this means that the impact factors are similar. However, this does not mean that the *effect* of these factors — either positive or negative — is equal in both projects.
- The set of impact factors themselves do not overlap significantly between projects.

Product-related factors
Resource constraints
– Timing
– Memory utilisation
– CPU occupancy
– Number of resource constraints
Program complexity
Client interface
– Experience
– Participation
Size of programming product

Project-related factors
Hardware development concurrent with programming
Development computer size
Requirements specification
– Client vs. ITT-written specification
– Amount of requirements rewritten
Modern programming practices usage
Personnel experience

Table 2: Factors that impact productivity (adapted from [VCW+84]).

Influence factors, team capabilities and experience
Programming language experience
Application experience and familiarity
Project manager experience and skills
Influence factors, software complexity
Database size and complexity
Architecture complexity
Complexity of interface to other systems
Influence factors, project constraints
Schedule pressure
Decentralized/multi-site development
Influence factors, tool usage and quality/effectiveness
CASE tools
Testing tools
Context factors
Programming language
Domain
Development type

Table 3: Factors that impact productivity [TM09].

If impact factors for a given project are known, it is possible to artificially increase or dampen productivity ratings using coefficients, making cross-project comparisons possible. Let us consider a theoretical scenario where two software engineering projects have the exact same context factors and influence factors. Let us further assume that one of these influence factors is the degree of customer participation in the project: the customer for project A does not participate directly in the project, but the customer for project B is highly involved in the decision-making process. The impact of all other influence factors and context factors is the same across projects. After calculating productivity ratings for both projects using LoC-based measurement, project A is found to have written 20 per cent more code during the same time as project B. Thus, customer participation has accounted for a 20 per cent increase in productivity. This degree of impact can be factored out either by decreasing the average productivity rating of project A by multiplying by a coefficient of $\frac{1}{1.2}$, or increasing the average productivity rating of project B by multiplying by a coefficient of 1.20.

The ability to devise appropriate coefficients entails not only knowing the impact factors for a given project, but also how much a given factor has impacted productivity inside the project. This can be achieved, for example, by asking personnel to identify impact factors for a project and assign a rating for each one. Another, decidedly more accurate approach is estimation based on a large sample of historical software projects. The Experience database, developed by Maxwell and Forselius, is a database containing function point-based productivity data for 206 software projects from 26 Finnish companies, divided into five major business sectors (banking, insurance, manufacturing, wholesale-retail and public administration) [MF00]. For all 206 submitted projects, key productivity factors have been identified and assigned an impact rating (very low to very high) based on a set of criteria developed for each factor. Thus, it is possible to extract estimated coefficients for any key factor in any of the five aforementioned business sectors, making cross-project comparisons easier. Additionally, such a database can be used to estimate the productivity in a new project and to improve productivity identifying which

impact factors typically have a negative affect on productivity (depending on a project's business sector).

Even when done with care, cross-project productivity analysis is based on estimates in the sense that establishing all of the impact factors of a project is not possible in practice. Ideally, cross-projects comparisons should only be carried out for projects in which not only notable impact factors are similar, but also their effect.

5 Established productivity metrics in software engineering

All software engineering — choice of development process notwithstanding — is made up of different activities. Such activities include elicitation of system requirements, elicitation of software requirements, analysis, program design, coding, testing and operations [Roy70]. These activities exist regardless of the chosen software development process. Due to the different nature of these activities, the output of a software project does not consist of only one type of unit, but several. Some established productivity metrics focus on one particular activity and thus one type of output unit, whereas others are designed to be applicable for all activities.

5.1 Productivity based on lines of code

Measurement based on lines of code is currently one of the most popular ways to gauge coding activity. In this model, source lines of code serves as the output and time spent programming (typically presented as person-months, *pm*) as the input or effort. Figure 2 presents a mathematical notation for the model.

$$f(sloc, pm) = \frac{sloc}{pm}, sloc \in \mathbb{N} \land pm \in \mathbb{R}^{+}$$

Figure 2: A model for productivity measurement based on LoC and person-months.

The primary benefits of this model are easy to identify. Firstly, the data required to use this model is easy to collect. Program code is typically stored in a code repository, from which it is simple to extract the lines of code written by a given team member, allowing the assessment of not only the entire programming team, but also individuals. Programming time is also easily collected, and can be approximated based on the length of of a project and known working hours. The simplicity of data collection is perhaps the main reason why LoC-based productivity measurement has managed to gain such a foothold in the software engineering industry.

Secondly, even without taking into account the validity criteria outlined in Chapter 3.2, it is clear the LoC-based productivity measurement does, to some extent, serve as an indicator of how productive a team member has been when programming: a programmer who has produced one thousand lines of code during one person-month is — from a programming perspective — more productive than a programmer who fails to produce any code at all during one person-month. It is when both programmers have produced some code during a given amount of time that such a distinction is difficult to make due to a host of major drawbacks with the model.

The primary drawback with LoC-based measurement is that since the metric is based solely on produced source code, it places no value in activities that do not result in code. In software projects, significant time can be spent on non-programming work. Such work, though not resulting in code, may still constitute a significant project contribution — forgoing the measurement of these types of contributions can be misleading.

Another issue inherent to using source code for productivity measurement is that programming languages differ from one another, both in terms of syntax and overall design. In a theoretical scenario where one programmer implements N features in one person month using the Ruby programming language, his or her LoC per person-month ration is likely to be lower than a programmer in the exact same scenario and context who implements N features in one person-month using the Java programming language, because Java is typically regarded as more verbose language than Ruby. This can result in erroneous values when using LoC-based metrics.

The lack of differentiation between levels of complexity is another — albeit less significant — problem with LoC-based measurement. Implementing a simple feature may well result in more lines of code than when implementing a complex one, despite the fact that both require the same amount of effort. Such circumstances may yield misleading results, interpretations, or both.

Given its dependence on programming language, LoC-based measurement must also contend with potential issues caused by language *idioms*. Idioms are encouraged ways to express popular language constructs; these idioms typically vary from language to language. Using idioms to implement an algorithm may require less code than implementing the same algorithm without the use of idioms. In such cases, using LoC-based measurement will result in higher productivity ratings for individuals who ignore language idioms, which is likely not a desired result.

In addition to using language idioms, different coding styles used within the same project may result in misleading productivity ratings. A developer who pays attention to code readability and understandability may write more verbose code than others, but may take a longer time to do so. Consider a scenario in which two developers, A and B, implement the same piece of application functionality. The code written by A is highly understandable, whereas the code written by B is equal in length yet much harder to grasp. If B produced his or her code faster than A, he or she would be considered more productive than A even though A placed more thought on proper code style.

Enforcing a common code style within a development project may mitigate the effects such scenarios have on LoC-based productivity measurement, but are unlikely to eliminate them entirely.

If one chooses to apply the LoC model despite the drawbacks presented above, deciding which parts of code should be used in counting a productivity rating is important and can lead to the incomparability of results when not clearly defined [Pet11]. Code that should generally not be counted includes the code found in third-party frameworks and libraries, as these may include vast amounts of functions or methods never actually used in the application-under-development. Failing to factor out framework or library code results, yet again, in misleading productivity ratings. Conversely, though framework and library code should not be counted, the use of such resources promotes reuse and may reduce the amount of defects found in application code [MCKS04]. A programmer unfamiliar with a library may spend more time implementing an algorithm using it and producing less own code than a programmer who writes the algorithm without the use of libraries. In such cases, LoC-based measurement always favours the individual who does not make use of external resources.

Lastly, the LoC productivity model — as well as all other productivity models — may be subject to intentional abuse by individuals subject to measurement. If programmers are aware that their productivity is being gauged based on lines of code written per time, it is possible to arbitrarily increase one's productivity rating by purposely writing more verbose code, adding unnecessary comments and inserting more line breaks. Such situations are, however, more a matter of professional integrity than an inherent flaw in the model.

The LoC model is still a popular productivity model today, even though many of the faults with the model are recognised. It is assumed that although flawed, there exists no better alternative [vV00].

5.2 Productivity based on function points

The concept of *Function Point Analysis* (FPA) was first introduced by Albrecht in 1979, primarily as a response to the fact that code length is subject to widely different values depending on the programming language used [Alb79] and as such unsuitable for determining the amount of functionality a piece of software comprises [FP98]. FPA can be used to estimate or measure the functional size of a given application and, by extension, be used to gauge the productivity of the team implementing it. According to Longstreet, function point analysis can be applied to an entire development project (including phases such as requirements analysis), to an entire enhancement project (where an existing application is expanded) and to existing applications (where effort is spent on maintenance) [Lon04].

The main result of applying function point analysis is a so-called function point count — a numeric value describing the amount of functionality an application provides its user. Function point counts are typically calculated by focusing on the following five entities [vV00]:

- Number of input types (I). An input type refers to user input that changes underlying data structures of an application.
- Number of output types (O). An output type refers to data output by an application.
- Number of inquiry types (E). An inquiry type refers to forms of user input that control the execution of an application, but do not change the underlying data structures of the application.
- Number of logical internal files (L). A logical internal file is a file used and maintained internally by an application.
- Number of interfaces (F). Interfaces are mechanisms by which data can be output to — or shared with — another application.

Figure 3 presents a mathematical notation for measuring programmer productivity with function point analysis. Here, fp is the function point count. The unit of effort is person-months (pm), the same as when using LoC-based productivity measurement.

$$f(fp, pm) = \frac{fp}{pm}, fp \in \mathbb{Z} \wedge pm \in \mathbb{R}^{+}$$

Figure 3: A model for productivity measurement based on function points and person-months.

The primary benefit of function point analysis for productivity measurement is its independence of chosen programming language and thus also the amount of source code produced. Assuming the same context and assumptions, an application written in Java should have the same amount of function points as the same application written in Ruby, for example. The second largest benefit of FPA is that function points can be counted before implementation has begun, since documentation produced during the design and requirements elicitation phases of a project should contain enough information about the application-under-development to identify transactions, files and general system characteristics.

The reliance on documentation enables the estimation of function points before implementation, but when measuring productivity using FPA, this benefit is lost: as such a measurement entails calculating how many function points were *implemented* during a period of time, the source code itself must be used as the basis for analysis.

This, in turn, leads to another issue: currently, function point analysis is best carried out manually. Though methods for automatic function point analysis exist [FTB06, HA99], counting function points requires the ability to recognise many different forms of inputs, outputs and patterns. Deciding when to award a function point is often subjective, and thus difficult behaviour for a machine to mimic. As a result, function point analysis software often requires interaction from the user when deciding when to award function

points. According to the International Function Point Users Group (IFPUG), such applications can be certified as Type 2 FPA software [ifp13]. The lack of Type 3 (fully automated) software is perhaps the most significant drawback that function point analysis has against the easily automated process of LoC-based productivity measurement.

5.3 Miscellaneous artefact-based productivity metrics

To measure productivity for development phases other than implementation, one must use artefacts other than source code as the basis for analysis. Such artefacts include requirements documentation, design documentation, test cases and user stories. A naïve approach to evaluating productivity for non-implementation phases would be to simply total the number of appropriate artefacts produced during a given period of time, e.g. the number of requirements documented per person-month as a metric for productivity during the requirements elicitation phase. Why such metrics are not commonly used is evident: not all projects produce the same types of artefacts and, even if many artefacts are present in most software engineering projects, they can differ wildly in scope and structure.

The challenge in devising metrics for non-implementation phases of software projects has proven so difficult that LoC-based measurement is often applied to entire projects, further compounding the metric's known issues. A metric model that is independent of produced artefacts, programming languages and phases of development is needed.

5.4 Productivity based on project velocity

Though the notion of a *task* is present in most forms of software development, it is an integral and systematic part of Agile software processes such as eXtreme Programming (XP) [Bec99], Scrum [Sza13] and scheduling systems

such as Kanban [IPF+11]. The exact definition of a task varies from one methodology to another, but in basic terms, a task can be defined as an assignment that is undertaken by one or more team members and marked as done when it is completed. In the Scrum process, for example, tasks manifest themselves as items in the *Sprint backlog* — a collection of planned work for the current Sprint[2] — and are decompositions of larger units of work known as *Product Backlog Items* [Sza07]. When using Kanban, tasks manifest themselves as tickets (index cards) on the Kanban Board.

Velocity can be determined for projects in which tasks are tracked and their effort estimated. Effort can be denoted using a common unit of measurement such as ideal engineering hours or days, or an abstract unit such as story points or t-shirt sizes. Given a set S of tasks and their total estimated effort e during an interval of time t, velocity is calculated by dividing the effort e with the interval t (Figure 4). In both XP and Scrum, it is recommended that t be set to the length of one development iteration [Wel12, Sza07].

$$f(e,t) = \frac{e}{t}, e \in \mathbb{N} \wedge t \in \mathbb{R}^+$$

Figure 4: A model for productivity measurement based on velocity.

Though commonly used as a mechanism to plan projects by determining how much work a development team can handle in a given iteration [Wel12, Sza07], velocity does serve as an indicator of productivity as it adheres to the definition of productivity (the ratio between what is produced and to the time required to produce it). In theory, velocity combines the benefits of LoC- and function point-based productivity metrics into a single model without also incorporating the drawbacks of these metrics. The primary benefit of velocity is that, like function point analysis but unlike LoC-based measurement, the model is independent of chosen programming language in the sense that source code is not an used as the model's input. The multitude of issues with lines of code as a basis for measurement are thus avoided.

[2]A *Sprint* can be defined as an iteration of work during which some increment of a product's functionality is implemented [Sza07].

In addition to being independent of programming language, velocity is in fact independent of any specific software project artefact. Tasks, though each associated with some type of software engineering activity, are counted in the same fashion regardless of what artefact (e.g. program code, design documentation, test cases) was produced upon completion. By performing velocity for different types of development activities, it is possible to measure productivity not only for programming, but also requirements elicitation, design, documentation etcetera — a significant advantage that neither function point analysis nor LoC-based measurement can offer. For example, a team whose assignment is to design but not implement a piece of software can be measured for design productivity based on how many design-related tasks they complete during a period of time — trying to measure design productivity using LoC-based measurement will inevitably result in a productivity rating of zero, as no code is produced.

Another major benefit of velocity is that — like LoC-based measurement but unlike function point analysis — data can be collected with little to no extra effort. For all well-known Agile development processes, software exists with which tasks (or their equivalent counterparts) can be created and tracked, providing detailed information on who worked on a task, when the task was finished and what type of task is in question. In effect, velocity using appropriate software allows one to automate the entire data collection process.

Despite not having any of the major drawbacks found in function point- or LoC-based productivity models, velocity is at a potential disadvantage in terms of the sizing or *scoping* of tasks. As tasks must be estimated in terms of the effort required to complete them, the underlying metric is only as accurate as the estimation itself. Estimations can be done by consulting members of the development team using an approach such as Planning Poker[3], or by assigning the estimation solely to an experienced project lead or manager.

[3]Planning Poker is an estimation technique where developers individually estimate the time it takes to implement a set of features before arriving at a consensus for the entire group.

6 Implicitly estimated velocity

Implicitly estimated velocity (IEV) is the model proposed in this work as a replacement for LoC-based productivity measurement. It entails counting the number of tasks an individual has performed during a given span of time. Though quite similar to measuring velocity, the key difference between velocity and implicitly estimated velocity is that the latter does not employ estimation of any sort: as mentioned in Chapter 5.4, a model that uses estimation is only as accurate as the estimation itself. Studies have indicated that estimates can vary greatly, especially in the beginning of a project [Abr03], thus lowering accuracy. For inexperienced developers, the problem is exacerbated as estimation is a skill that improves over time [Hum95]. The implicitly estimated velocity approach suggests that by forgoing estimates, one will still end up with a valid model. This is based on the assumption that even without explicit estimation, all projects that base development around the notion of a task do, in fact, *implicitly* estimate tasks.

Let us consider a scenario in which Agile and Lean methodologies are used in a development project, but no explicit estimation is done. The implicitly estimated velocity model assumes the following:

- When any task is being defined, the overall context, domain and make-up of the development team dictates that there is an implicit lower and upper size or effort limit within which the task must lie before it is accepted as a task.
- As time goes on, the team's ways of working narrow this possible size interval.
- The likelihood of a team member choosing or being assigned only comparatively small tasks, thus misleadingly achieving a high task count, is mitigated when adopting methodologies that advocate the use of task prioritisation and work-in-progress limits.

- The likelihood of a team member choosing or being assigned only comparatively small tasks is further mitigated in cases where the individual responsible for defining a task is not the same individual who is assigned the task.

Figure 5 describes a model for measuring productivity using implicitly estimated velocity. The unit of effort pm is, as with LoC-based measurement, person-months. The task count, tc, is defined as the total number of tasks completed by an individual or, if one wishes to measure an entire development team, the total number of tasks completed by the all development team members.

$$f(tc, pm) = \frac{tc}{pm}, tc \in \mathbb{N} \land pm \in \mathbb{R}^+$$

Figure 5: A model for productivity measurement based on implicitly estimated velocity and person-months

In theory, implicitly estimated velocity shares all of the benefits of productivity measurement using velocity: the model is both independent of any programming language and easy to use in practice, especially if a development team is using an electronic task tracking system. Even when using physical Kanban Boards, as is sometimes the case in co-located Agile and Lean projects, it is relatively easy to periodically collect and count completed task notes.

Another benefit is that by forgoing the estimation of tasks, the time spent estimating how long a task takes to complete is freed and thus be used for other activities. This reduces waste, one of the core principles of Lean software development [PP03]. Specifically, it reduces waste due to unnecessary delays whilst developing software.

7 Case study design

A case study can be characterised as method focused on investigating contemporary phenomena in their specific contexts [RH09]. There is no single definition as to what constitutes a case study. According to Robson, a case study is a research strategy that makes use of multiple sources of evidence [Rob02]. According to Yin, it is an inquiry in which the boundary between the phenomenon under study and its context may not be clear [Yin03].

The goal of the case study in this work is to address formal research question based on the general questions posed in Chapter 1. These questions are:

1. Is productivity based on lines of code a valid metric for productivity in Agile and Lean software development?
2. Is the proposed concept of implicitly estimated velocity a valid metric for productivity in Agile and Lean software development?
3. If productivity based on lines of code is unsuitable in an Agile and Lean environment, could implicitly estimated velocity serve as a viable replacement?

The case study is comprised of development projects carried out at the Software Factory. As previously mentioned, the main focus is on the established method of LoC-based measurement and the proposed concept of implicitly estimated velocity.

In general, the case study follows the linear-analytical approach described by Yin [Yin03]. This section describes the overall context of the Software Factory, the research approach (including formal research questions and methodology), the data collection process and threats to validity. Chapters 8 and 9 provide a detailed analysis of both projects under study.

7.1 General context

The Software Factory (henceforth referred to as the *Factory*), part of the University of Helsinki's Computer Science department, was established in January 2010 and serves as an experimental research and development laboratory and academic effort in which MSc-level student teams develop working prototypes of applications during a full-time span of approximately seven weeks, with each work day consisting of six hours. Students do not receive any monetary compensation for their efforts; instead, they are awarded credits based on how many days per week they work: ten ECTS credits are awarded for four-day work weeks, twelve ECTS credits for five-day work weeks.

The Factory located in Helsinki is part of a larger, global initiative: currently, similar Software Factory laboratories exist elsewhere in Finland (Joensuu and Oulu) and Europe (Bolzano, Italy; Madrid, Spain) [Koi13].

All projects undertaken at the Factory are "commissioned" by industry organisations; as such, there is always a real business demand behind each project, adding credence to the validity of the Factory as a test bed for research. Researchers from both the University of Helsinki and outside institutions are welcomed to perform any type of research in the Factory. Possible research methods include surveys, interviews, direct observations, action research and data mining.

Regardless of the project, the Factory favours Lean software development by adopting Scrumban as its primary development process. In essence, Scrumban is an amalgamation of the Agile software development process Scrum and the Kanban scheduling system. For the projects under study, a physical Kanban Board was used in order to track tickets or tasks.

7.2 Research questions

The general questions posed at the beginning of this work must be refined into formal research questions that conform to the constraints of the case study. This entails taking the overall context of the case study into account. By doing so, the original questions can be refined into candidates for research questions (CRQs) as follows:

- **Q1**: Is productivity based on lines of code a valid metric for productivity in Agile and Lean software development?
  - **CRQ1**: Is LoC-based measurement a valid productivity metric in the Agile and Lean software development process used in the Software Factory?
- **Q2**: Is the proposed concept of implicitly estimated velocity a valid metric for productivity in Agile and Lean software development?
  - **CRQ2**: Is the proposed concept of implicitly estimated velocity a valid productivity metric in the Agile and Lean software development process used in the Software Factory?
- **Q3**: If productivity based on lines of code is unsuitable in an Agile and Lean environment, could implicitly estimated velocity serve as a viable replacement?
  - **CRQ3**: Is implicitly estimated velocity a more valid productivity metric than LoC-based measurement in the Agile and Lean software development process used in the Software Factory?

As previously mentioned, the validity of a metric depends on what thresholds are deemed suitable. For the purposes of this study, there is no need to initially regard one criterion more important than another — determining for which criteria a metric fares poorly and thus risks being deemed invalid yields

a more interesting discussion. Applying this logic to the candidate research questions results in three final research questions:

- **RQ1**: Under which circumstances may LoC-based measurement be deemed an invalid productivity metric in the Agile and Lean software development process used in the Software Factory projects under study?
- **RQ2**: Under which circumstances may implicitly estimated velocity be deemed an invalid productivity metric in the Agile and Lean software development process used in the Software Factory projects under study?
- **RQ3**: Under which circumstances is implicitly estimated velocity a more valid productivity metric than LoC-based measurement in the Agile and Lean software development process used in the Software Factory projects under study?

7.3 Data collection

In order to analyse the validity of productivity measurement using LoC, the code base produced during a project must be inspected. Both projects examined in this case study used the Git version control system to store code. Using a version control system greatly simplifies LoC-based measurement: given a properly configured environment, both collective and individual lines of code written can be extracted from the underlying repository. As framework code can lead to misleading LoC counts, it has been omitted from the results of the study[4].

As the Factory projects made use of a physical Kanban board, task data was collected manually. Before the start of both projects, teams were instructed to denote each task with the current date and time when completed, along with the names of the team members that worked on the task in question.

[4]Repository commits consisting of over 100 line insertions were considered framework code; such a low threshold was chosen to eliminate as many false positives as possible.

For any given task worked on by two or more team members, individual effort is considered equal (one task per participating member). Though this may be slightly misleading, it was not possible to determine participation in a given task more accurately.

To analyse both LoC-based measurement and implicitly estimated velocity against the validity criteria, an additional data set describing the attribute in question (productivity) is required. In this study, the expert opinion of the project team members themselves was chosen as such a data set. For both projects, each team member was posed two questions after their respective projects had ended:

1. *How productive do you believe each team member was in terms of programming (coding) during the project? Also assess yourself.*
2. *How productive do you believe each team member was in general during the project? Also assess yourself.*

Team members answered both questions by rating their peers on an interval scale from one to ten, one being "not productive at all" and ten "highly productive". Answers were kept private — no team member was able to see the ratings given to others. Though individuals assessed themselves as well as others, self-assessments have been omitted from this study to prevent bias. For practical purposes, the questions were included as part of a larger, end-of-project online questionnaire for each respective project. Prior to filling out the questionnaire, all team members were introduced to the definition of productivity as it appears in this thesis. In particular, students were instructed to take variations in working days into account and not to consider permitted absent days as having a negative impact on an individuals productivity.

In order to accurately determine the extent to which a metric fulfills the tracking criterion (outlined in Chapter 3.2), multiple data points taken at different points in time are needed. Whilst this data has been collected for

the metrics under study, the questionnaire was only answered once by each team member. Thus, it is unfortunately not possible to examine LoC-based measurement and implicitly estimated velocity against the tracking criterion in this study.

All of the data collection performed in this case study can be classified as *second-degree collection*, as subjects were not interacted with directly [RH09]. Respecting privacy, all data has been anonymised so that no single person can be distinguished from any of the results.

7.4 Limitations and threats to validity

In any case study, a listing of possible threats to the validity of the study should be present so that the reader can determine to which extent the results are valid and extendible outside the scope of the study. The following classification scheme is used (adapted from Yin [RH09]):

- *Construct validity* describes to what extent operational measures under study represent what the case study researcher intended them to represent. Intuitively, construct validity describes to what degree operational measures really have to do with the formal research questions of a study.
- *Internal validity* is only of concern if causal relationships are examined. If a factor f_1 is believed to affect another factor f_2, one must be aware of any additional factors that might affect f_2.
- *External validity* describes to what extent it is possible to generalise any findings and to what extent the findings may be of interest to people outside of the case study. Though statistically significant results are typically not found during a case study, results may be extended to other studies with common characteristics.
- *Reliability* describes to what extent the collected data and analysis thereof are dependent on the researcher(s) conducting the study. Hypo-

thetically, if another researcher were to conduct the exact same study, the results should be identical.

The construct validity of the productivity metrics under study is not threatened as the purpose of this case study is to determine if these metrics do, in fact, accurately measure productivity. Thus, no assumptions have been made as to what these constructs represent.

For the two questions in which project participants were asked to assess the productivity of their team members, construct validity may be threatened if team members collectively decide to award each other a high productivity rating, regardless of how productive a team member actually was. Incorporating the study's questionnaire as part of the course's "official" end-of-project questionnaire may have mitigated this risk, as the questionnaire is used in part to determine the grades students' received for completing their respective projects and thus likely answered truthfully. However, it is not possible to determine if this is indeed the case.

The internal validity of this case study is not threatened as causal relationships are neither suggested nor examined. The external validity, however, is subject to a number of limitations. Results of a case study can typically not be generalised [RH09, SCC02]. Nevertheless, the results of this case study do form one example of using implicitly estimated velocity in an Agile and Lean environment — enough upon which to form a theory for continued research on the subject.

Regarding the reliability of the study, all data collection methods have been explained in detail in this chapter, rendering an accurate recreation of the results possible. Including all task data in this thesis is omitted due to the sheer size of the data set, but are available upon request.

8 Project 1

The first project under study took place between September 3, 2012 and October 19, 2012, for a total of 35 working days (not counting weekends nor holidays). During this project, a metrics and analytics front-end for an embedded Internet telephony system was designed and developed. The primary programming language used was Java.

A total of four students participated in development, with one student electing to leave mid-project. Contributions of this team member have been omitted from this analysis. The length of a person-month has been defined as $35 \times 6 = 210$ hours.

8.1 Analysis of programming productivity

This chapter examines the correlation, predictability and discriminative power of both LoC-based measurement and implicitly estimated velocity for programming productivity measurement in Project 1.

8.1.1 Correlation

Table 4 describes the total lines of code committed by each member of the team, along with their respective LoC-based measurement ratings and reported total working hours. A total of 9952 LoC were written, with team member B writing a significant percentage — 46 per cent — of the entire application, thus being the most productive member according to this metric.

Before a fair comparison with other team members can be carried out and a proper correlation coefficient calculated, the productivity ratings of each team member must be adjusted against a person-month length of 210 hours. Table 5 presents the revised productivity ratings for each team member.

Team member	**LoC written**	**Reported hours**	**Rating ($\frac{loc}{pm}$)**
A	3887	210	$\frac{3887}{pm}$
B	4560	177	$\frac{4560}{0.842pm}$
C	1505	169	$\frac{1505}{0.804pm}$
Total	9952	556	

Table 4: LoC-based productivity ratings for Project 1.

Team member	**LoC written**	**Rating ($\frac{loc}{pm}$)**
A	3887	$\frac{3887}{pm}$
B	4560	$\frac{5410}{pm}$
C	1505	$\frac{1870}{pm}$
Total	9952	

Table 5: Adjusted LoC-based productivity ratings for Project 1.

Ratings calculated using the LoC-based metric do not exactly mirror that of individual team members' assessment of each others' programming productivity. Table 6 outlines how each team member rated both their own and other's programming productivity. By calculating the mean of all ratings, A and B were regarded as the most productive, despite A having written considerably more code than B. Calculating the correlation between the adjusted productivity ratings and the means of the peer-assessment does, however, yield a strong positive correlation coefficient of 0.903 for LoC-based measurement.

In order to measure programming productivity using implicitly estimated velocity, one can simply omit tasks that are not strict programming tasks before applying the metric. Table 7 presents the number of programming tasks completed per team member, along with their rating.

As with LoC-based measurement, the ratings must be normalised for a person-month of 210 hours before comparisons can be made. Table 8 describes the adjusted programming productivity ratings for each team member.

	A	**B**	**C**
A	-	8	7
B	9	-	8
C	8	9	-
Mean	$8\frac{1}{2}$	$8\frac{1}{2}$	$7\frac{1}{2}$

Table 6: Team members' assessment of programming productivity in Project 1.

Team member	**Tasks completed**	**Reported hours**	**Rating** ($\frac{tc}{pm}$)
A	32	210	$\frac{32}{pm}$
B	24	177	$\frac{24}{0.842pm}$
C	22	169	$\frac{22}{0.804pm}$
Total	78		

Table 7: Programming tasks completed in Project 1.

Team member	**Tasks completed**	**Rating** ($\frac{tc}{pm}$)
A	32	$\frac{32}{pm}$
B	24	$\frac{28.5}{pm}$
C	22	$\frac{27.3}{pm}$

Table 8: Adjusted programming productivity ratings for Project 1.

Correlating these adjusted productivity ratings with the results of the peer-assessment gives a coefficient of 0.697 — a lower correlation than when using LoC-based measurement. Thus, for programming work in this project, the correlation criterion is better fulfilled when using LoC-based measurement over implicitly estimated velocity. The correlation coefficient for implicitly estimated velocity does, however, indicate a moderate positive association against team members' perception of productivity.

8.1.2 Consistency

Calculating the rank correlation between team members' assessment of programming productivity and the metric results yields values of 0.866 for both LoC-based measurement and implicitly estimated velocity. Thus, for programming productivity in Project 1, both metrics can be considered equally consistent.

8.1.3 Predictability

Table 9 presents the coefficients of variation for both lines of code committed per day and programming tasks completed per day, for both the entire team as well as individual team members. A lower coefficient of variation is better, indicating less variation in work done from day to day.

	LoC-based measurement	**IEV**
Entire team	**1.468**	1.566
Team member A	2.680	**1.686**
Team member B	2.354	**2.145**
Team member C	1.973	**1.686**

Table 9: Coefficients of variation for programming productivity in Project 1.

When assessing the entire team, LoC-based measurement provides a somewhat higher degree of predictability in this project than implicitly estimated velocity. However, when assessing individual team members, implicitly estimated velocity provides a greater degree of predictability in all three cases, as indicated by lower coefficients of variation.

8.1.4 Discriminative power

Using the total lines of code written per person and programming tasks completed per person, coefficients of variation can be used to assess the discriminative power of LoC-based measurement and implicitly estimated velocity with respect to programming productivity.

As shown in Table 5, team members A, B and C wrote 3887, 5410 and 1870 lines of code, adjusted for their total working hours. In terms of programming tasks, the figures are 32, 28.5 and 27.3 tasks, respectively (Table 8). Calculating the coefficient of variation for each of these two series gives 0.477 for LoC-based measurement and 0.083 for implicitly estimated velocity. The coefficient for LoC-based measurement is considerably higher, indicating that the discriminative power of this metric is higher than implicitly estimated velocity when assessing programming productivity in this project.

8.2 Analysis of general productivity

This chapter examines the correlation, predictability and discriminative power of both LoC-based measurement and implicitly estimated velocity for general productivity measurement in Project 1.

8.2.1 Correlation

Calculating general productivity using implicitly estimated velocity entails simply counting the total number of tasks completed per team member, without omitting any specific type of task or group of tasks.

Table 10 describes the total number of tasks completed by each team member. A total of 219 tasks were completed during the project, with almost half contributed to by team member A (circa 42 per cent). Team members B and C contributed to roughly the same percentage of all tasks — 31 and 29 per

cent, respectively.

Team member	**Tasks completed**	**Reported hours**	**Rating** ($\frac{tc}{pm}$)
A	93	210	$\frac{93}{pm}$
B	66	177	$\frac{65}{0.842pm}$
C	60	169	$\frac{60}{0.804pm}$
Total	219		

Table 10: General tasks completed in Project 1.

Adjusting team members' output to a 210-hour person-month week yields ratings of 93, 78.3 and 74.6 tasks per month for members A, B and C (see Table 11). Even with this adjustment, implicitly estimated velocity indicates that A was the most productive team member during the project.

Team member	**Tasks completed**	**Rating** ($\frac{tc}{pm}$)
A	93	$\frac{93}{pm}$
B	66	$\frac{78.3}{pm}$
C	60	$\frac{74.6}{pm}$
Total	219	

Table 11: Adjusted general productivity ratings for Project 1.

The general productivity ratings provided by the implicitly estimated velocity method are reflected well by team members' assessment of each other. Table 12 presents the mean of the general productivity grades given to each team member. A was regarded as the most productive, followed by B and then C — the same outcome as indicated by implicitly estimated velocity. Correlating the productivity rating of each team member with their respective mean grades gives a coefficient of circa 0.945.

As mentioned earlier, LoC-based measurement is based on source code and thus only measures programming productivity, not the general productivity of any given software project. It is, however, sometimes applied to entire

	A	**B**	**C**
A	-	9	8
B	9	-	8
C	9	8	-
Mean	9	$8\frac{1}{2}$	8

Table 12: Team members' assessment of general productivity in Project 1.

projects. Doing so in this project results in a correlation that is noticeably lower than when using implicitly estimated velocity. Using the adjusted LoC-based ratings from Table 5, we receive a correlation coefficient of 0.568. In terms of correlation, it is clear that general productivity is better served by implicitly estimated velocity.

8.2.2 Consistency

Calculating the rank correlation between team members' assessment of general productivity and the metric results yields values of 0.5 and 1.0 for LoC-based measurement and implicitly estimated velocity, respectively. For Project 1, it is clear that implicitly estimated velocity is able to rank the team members with regards to general productivity better than LoC-based measurement. Thus, the former is more consistent.

8.2.3 Predictability

Table 13 presents the coefficients of variation for both lines of code committed per day and tasks completed per day, for both the entire team as well as individual team members. As the LoC-based measurement metric is based on lines of code written, the coefficients of variation calculated for this metric are the same as for programming productivity.

	LoC-based measurement	**IEV**
Entire team	1.468	**0.951**
Team member A	2.680	**0.995**
Team member B	2.354	**1.184**
Team member C	1.973	**1.742**

Table 13: Coefficients of variation for general productivity in Project 1.

In contrast to programming productivity, general productivity ratings in this project are more predictable using implicitly estimated velocity over LoC-based measurement, both when assessing the team as a whole and individual team members. In both cases, the difference in coefficients is significant.

8.2.4 Discriminative power

The total number of tasks completed per team member — 93, 78.3 and 74.6 for A, B and C, respectively — gives a coefficient of variation of 0.119 for the discriminative power of general productivity measurement using implicitly estimated velocity. Using LoC-based measurement, the discriminative power is the same as when assessing programming productivity (0.477). Clearly, the discriminative power of LoC-based measurement is higher than that of implicitly estimated velocity when assessing general productivity in this project.

9 Project 2

The second project under study took place between October 29, 2012 and December 14, 2012, for a total of 35 working days. During this project, the metrics and analytics front-end developed during Project 1 was extensively tested and further developed to handle a significant increase in concurrent

connections and stored data. The primary programming languages used were JavaScript (Node.js) and Python.

A total of five students participated in development, with one student electing to leave mid-project. Contributions of this team member have been omitted from this analysis. Given 35 working days, the length of a person-months has been defined as $35 \times 6 = 210$ hours — the same as for Project 1.

9.1 Analysis of programming productivity

This chapter examines the correlation, predictability and discriminative power of both LoC-based measurement and implicitly estimated velocity for programming productivity measurement in Project 2.

9.1.1 Correlation

Table 14 describes the total lines of code committed by each member of the team, along with their respective LoC-based measurement ratings and reported total working hours. A total of 9479 LoC were written during this project.

Team member	**LoC written**	**Reported hours**	**Rating** ($\frac{loc}{pm}$)
A	1848	195	$\frac{1848}{0.929pm}$
B	1288	197	$\frac{1288}{0.938pm}$
C	1695	161	$\frac{1695}{0.767pm}$
D	4648	193	$\frac{4648}{0.919pm}$
Total	9479	746	

Table 14: LoC-based programming productivity ratings for Project 2.

Based on the actual lines of code written, team member D appears to be the most productive individual. Adjusting the ratings in accordance with a 210-hour person-month does not change the highest-ranked team member, but does alter the rankings of the remaining three members (Table 15).

Team member	**LoC written**	**Rating ($\frac{loc}{pm}$)**
A	1848	$\frac{1990}{pm}$
B	1288	$\frac{1372}{pm}$
C	1695	$\frac{2210}{pm}$
D	4648	$\frac{5057}{pm}$
Total	9479	

Table 15: Adjusted LoC-based programming productivity ratings for Project 2.

Using implicitly estimated velocity, programming productivity ratings differ from that of LoC-based measurement. Table 16 presents the number of programming tasks completed per team member, along with their initial rating. Adjusting these ratings for a 210-hour person-month results in team member C being the most productive, followed by A, B and D (Table 17).

Table 18 outlines how each team members rated both their own and other's programming productivity. Interestingly, the mean is equal for all team members, indicating that collectively, the team regarded each team member equally productive.

This result renders assessment against the correlation impossible: given no variation in mean averages, it is not possible to calculate a correlation coefficient for either LoC-based measurement or implicitly estimated velocity for programming productivity in this project.

Team member	**Tasks completed**	**Reported hours**	**Rating** ($\frac{tc}{pm}$)
A	16	195	$\frac{16}{0.929pm}$
B	9	197	$\frac{9}{0.938pm}$
C	15	161	$\frac{15}{0.767pm}$
D	8	193	$\frac{8}{0.919pm}$
Total	48	746	

Table 16: Programming tasks completed in Project 2.

Team member	**Tasks completed**	**Rating** ($\frac{tc}{pm}$)
A	16	$\frac{17.2}{pm}$
B	9	$\frac{9.6}{pm}$
C	15	$\frac{19.6}{pm}$
D	8	$\frac{8.7}{pm}$
Total	48	

Table 17: Adjusted programming productivity ratings for Project 2.

	A	**B**	**C**	**D**
A	-	10	10	9
B	10	-	10	10
C	10	10	-	10
D	9	9	9	-
Mean	$9\frac{2}{3}$	$9\frac{2}{3}$	$9\frac{2}{3}$	$9\frac{2}{3}$

Table 18: Team members' assessment of programming productivity in Project 2.

9.1.2 Consistency

Calculating the rank correlation between team members' assessment of programming productivity and the metric results is not possible due to the equal

means. As such, the result of the consistency inspection for programming productivity in Project 2 must be considered undefined. As such, it is not clear which of the metrics has better consistency in this case.

9.1.3 Predictability

Table 19 presents the coefficients of variation for the lines of code committed per day and programming tasks completed per day, both for the whole team as well as individual team members. For all but one team member, the figures favour LoC-based measurement as a more predictable metric for programming productivity in this project.

	LoC-based measurement	**IEV**
Entire team	**1.456**	1.832
Team member A	**1.574**	1.787
Team member B	**1.705**	2.180
Team member C	1.393	**0.655**
Team member D	**2.447**	3.198

Table 19: Coefficients of variation for programming productivity in Project 2.

9.1.4 Discriminative power

Given LoC counts of 1990, 1372, 2210 and 5057 for team members A, B, C and D respectively, the series of total lines of code written per team member gives a coefficient of variation of 0.617. With implicitly estimated velocity, the respective series is 17.2, 9.6, 19.6 and 8.7 tasks completed, for a coefficient of variation of 0.395. As with Project 1, measuring programming productivity using LoC-based measurement provides a higher degree of discriminative power than implicitly estimated velocity.

9.2 Analysis of general productivity

This chapter examines the correlation, predictability and discriminative power of both LoC-based measurement and implicitly estimated velocity for general productivity measurement in Project 2.

9.2.1 Correlation

Table 20 describes the total number of tasks completed by each team member. A total of 107 tasks were completed during the project. Team member A completed the most tasks — approximately 34 per cent of the entire team's output — followed by C, D and B. Adjusting team members' output based on their reported hours does not change this order (Table 21).

Team member	**Tasks completed**	**Reported hours**	**Rating** ($\frac{tc}{pm}$)
A	36	195	$\frac{36}{0.929pm}$
B	20	197	$\frac{20}{0.938pm}$
C	28	161	$\frac{28}{0.767pm}$
D	23	193	$\frac{23}{0.919pm}$
Total	107	746	

Table 20: General tasks completed in Project 2.

Team member	**Tasks completed**	**Rating** ($\frac{tc}{pm}$)
A	36	$\frac{38.8}{pm}$
B	20	$\frac{21.3}{pm}$
C	28	$\frac{36.5}{pm}$
D	23	$\frac{25}{pm}$
Total	107	

Table 21: Adjusted general productivity ratings for Project 2.

Table 22 presents the general productivity grades given to each team member in Project 2. Correlating the means of the grades with the total tasks per team member gives a coefficient of 0.4206 for implicitly estimated velocity.

	A	**B**	**C**	**D**
A	-	10	10	8
B	10	-	10	10
C	10	10	-	10
D	9	9	9	-
Mean	$9\frac{2}{3}$	$9\frac{2}{3}$	$9\frac{2}{3}$	$9\frac{1}{3}$

Table 22: Team members' assessment of general productivity in Project 2.

Calculating the corresponding correlation for LoC-based measurement yields a negative coefficient of -0.976, indicating an *inverse* relationship between LoC written and team members' perception of general productivity. As in Project 1, implicitly estimated velocity fares better than LoC-based measurement in terms of correlation for general productivity.

9.2.2 Consistency

Calculating the rank correlation between team members' assessment of programming productivity and the metric results yields values of -0.775 and 0.258 for LoC-based measurement and implicitly estimated velocity, respectively. Thus, for general productivity in Project 1, implicitly estimated velocity is more consistent, though neither metric succeeds in ranking general productivity well enough to accurately reflect the team member assessments.

9.2.3 Predictability

Table 23 presents the coefficients of variation for both lines of code committed per day and tasks completed per day (no tasks omitted) in Project 2. When

assessing the entire team, implicitly estimated velocity is more predictable than LoC-based measurement, as indicated by a lower coefficient value. For individual team members, implicitly estimated velocity is more predictable in three of the possible four cases.

	LoC-based measurement	**IEV**
Entire team	1.456	**0.955**
Team member A	1.574	**1.190**
Team member B	**1.705**	1.765
Team member C	1.393	**0.994**
Team member D	2.447	**1.056**

Table 23: Coefficients of variation for general productivity in Project 2.

9.2.4 Discriminative power

The approximate total of tasks completed per team per team member — 38.8, 21.3, 36.5 and 25 for A, B, C, and D, respectively — gives a coefficient of variation of 0.282 for general productivity measurement using implicitly estimated velocity. For LoC-based measurement, the discriminative power is described by the same coefficient of variation found when assessing programming productivity: 0.617. As in Project 1, the discriminative power of LoC-based measurement is higher than that of implicitly estimated velocity when assessing team members' general productivity.

10 Results

Table 24 summarises the key findings of the case study. If we choose to accept all other criteria as valid and focus solely on correlation, we can determine which of the metrics under study more accurately reflects the expert opinions of development team members. If we were to fix the threshold

value to a positive association of 0.5, then LoC-based measurement is valid for programming productivity at a reliability level of 50 per cent, as is implicitly estimated velocity. Setting the threshold at strong positive association of 0.9 yields a valid result for LoC-based measurement at 50 per cent reliability; a valid result for implicitly estimated velocity at this threshold would require accepting a reliability percentage of zero. Intuitively, in terms of correlation, LoC-based measurement is more likely to be valid than implicitly estimated velocity for programming productivity in this study, depending on the chosen correlation threshold.

The opposite is true when discussing general productivity measurement. If we select a correlation threshold of either 0.5 or 0.9, implicitly estimated velocity is valid at a reliability level of 50 per cent. LoC-based measurement is only equally reliable at a threshold of 0.5.

Consistency is closely related to the correlation criterion — it too determines to which degree both metrics under study reflects the opinions of a development team members. However, consistency only concerns itself with ranking team members from most productive to least productive — not with the difference in work done between ranks. If a metric accurately reflects development team members' assessment of productivity, the consistency criterion will always yield a value of 1.0, even if the differences between team members' output are large or comparatively minute.

If consistency were the only criterion of significance, implicitly estimated velocity is more likely to be considered valid than LoC-based measurement for general productivity. In Project 1, implicitly estimated velocity accurately ranked team members' productivity to reflect assessments. In Project 2, the rankings differed from those of the assessment means, but the result was nevertheless better than for LoC-based measurement, which yielded a negative correlation coefficient. If we were to set a consistency threshold of 0.8, implicitly estimated velocity would be considered valid at a reliability level of 50 per cent, whereas LoC-based measurement would only be considered valid if zero per cent reliability were acceptable (or the threshold lowered).

Criterion	**Project**	**LoC-based**	**IEV**
Correlation, prog. prod.	1	**0.903**	0.697
Correlation, prog. prod.	2	undefined	undefined
Correlation, gen. prod.	1	0.568	**0.945**
Correlation, gen. prod.	2	-0.976	**0.421**
Consistency, prog. prod.	1	**0.866**	**0.866**
Consistency, prog. prod.	2	undefined	undefined
Consistency, gen. prod.	1	0.5	**1.0**
Consistency, gen. prod.	2	-0.775	**0.258**
Tracking	1	N/A	N/A
Tracking	2	N/A	N/A
Predictability, prog. prod., team	1	**1.468**	1.566
Predictability, prog. prod., indiv.	1	2.680	**1.686**
Predictability, prog. prod., indiv.	1	2.354	**2.145**
Predictability, prog. prod., indiv.	1	1.973	**1.686**
Predictability, prog. prod., team	2	**1.456**	1.832
Predictability, prog. prod., indiv.	2	**1.574**	1.787
Predictability, prog. prod., indiv.	2	**1.705**	2.180
Predictability, prog. prod., indiv.	2	1.393	**0.655**
Predictability, prog. prod., indiv.	2	**2.447**	3.198
Predictability, gen. prod., team	1	1.468	**0.951**
Predictability, gen. prod., indiv.	1	2.680	**0.995**
Predictability, gen. prod., indiv.	1	2.354	**1.184**
Predictability, gen. prod., indiv.	1	1.973	**1.742**
Predictability, gen. prod., team	2	1.456	**0.955**
Predictability, gen. prod., indiv.	2	1.574	**1.190**
Predictability, gen. prod., indiv.	2	**1.705**	1.765
Predictability, gen. prod., indiv.	2	1.393	**0.994**
Predictability, gen. prod., indiv.	2	2.447	**1.056**
Disc. power, prog. prod.	1	**0.477**	0.083
Disc. power, prog. prod.	2	**0.617**	0.395
Disc. power, gen. prod.	1	**0.477**	0.119
Disc. power, gen. prod.	2	**0.617**	0.282

Table 24: Summary of the case study.

For programming productivity, the results from the case study are somewhat inconclusive. In the first project, both LoC-based measurement and implicitly estimated velocity received a rank correlation coefficient of 0.866. In the second project, the result was undefined due to equal assessment means. Based on these results, we can conclude that both metrics are as likely to be considered valid (or invalid) if consistency were the only criterion of significance. If we were to set a threshold of 0.8, both metrics would be considered valid, but only at 50 per cent reliability.

The predictability of a productivity metric explains how accurately one can predict the future productivity of a development team or its members based on previous productivity data. Having a metric with a provenly high degree of predictability is of importance when one wishes to estimate the completion date of an ongoing project, or to determine how much work a team can be assigned in future development iterations.

In terms of predictability in programming productivity, the results from the case study are mixed. When the unit of analysis is the entire software team, LoC-based measurement fares better than implicitly estimated velocity. For individuals, implicitly estimated velocity offers better predictability in Project 1, but worse predictability in Project 2. Setting a predictability threshold of 2.0 would deem LoC-based measurement valid at a reliability level of 66.7 per cent, provided that all other criteria are considered satisfied. For implicitly estimated velocity, the same reliability level would also yield a valid outcome.

For general productivity, implicitly estimated velocity is clearly more predictable than LoC-based measurement. Spanning both projects, the coefficients of variations calculated were eight out of nine times in favour of it. If we were to set a predictability threshold of 1.0 and deem all other criteria satisfied, implicitly estimated velocity would be considered valid at a reliability level of 50 per cent, whereas LoC-based measurement would be valid only if zero per cent reliability is acceptable.

LoC-based measurement fares clearly better than implicitly estimated velocity in terms of its discriminative power — its ability to distinguish between good and poor productivity within the context of chosen project. This holds true not only for both projects under study, but also for both general productivity and programming productivity. If we were to set the threshold for discriminative power at 0.450 and disregard the thresholds of other criteria, LoC-based measurement would be considered a valid metric at a reliability level of 100 per cent, whereas implicitly estimated velocity would require a reliability level of zero per cent.

Though not analysed, measuring productivity in terms of non-programming work is by default better served by implicitly estimated velocity than LoC-based measurement: during non-programming activities, no source code is produced. The correlation criterion, for example, will always result in an undefined value for LoC-based measurement.

Based on the case study findings, the formal research questions posed in Chapter 7.2 can be answered as follows:

- **RQ1**: Under which circumstances may LoC-based measurement be deemed an invalid productivity metric in the Agile and Lean software development process used in the Software Factory projects under study?
  - LoC-based measurement may be deemed invalid in cases where general productivity is being measured and little to no importance is being placed on a metric's discriminative power.
- **RQ2**: Under which circumstances may implicitly estimated velocity be deemed an invalid productivity metric in the Agile and Lean software development process used in the Software Factory projects under study?
  - Implicitly estimated velocity may be deemed invalid in cases where programming productivity is being measured instead of general productivity and the discriminative power and correlation criteria of a metric are valued over its predictability.

- **RQ3**: Under which circumstances is implicitly estimated velocity a more valid productivity metric than LoC-based measurement in the Agile and Lean software development process used in the Software Factory projects under study?
  - If both metrics are considered valid, LoC-based measurement is more valid than implicitly estimated velocity for measuring programming productivity for all validity criteria except predictability, whose results are inconclusive. For general productivity measurement, however, implicitly estimated velocity is more valid than LoC-based measurement for all criteria except discriminative power.

The correlation criterion is the single most important validity criterion, as it describes the extent to which figures provided by a productivity metric are related to team members' expert opinion of each others' productivity. As such, it is difficult to imagine a scenario in which one would favour one valid metric over another if its correlation coefficient is lower. Summarising the findings of the study, it is clear that implicitly estimated velocity is more suitable for general productivity measurement than LoC-based measurement. For programming productivity, LoC-based measurement is the better option. The correlation criterion does, however, indicate a moderate positive association also for implicitly estimated velocity, indicating that the metric still measures what it purports to measure even when only programming activities are under inspection.

Intuitively, the findings of the case study suggest that implicitly estimated velocity paints a more complete picture of a software teams' productivity than LoC-based measurement, a metric that is designed around the notion that source code is the only artefact of significance in software development.

11 Conclusions

Software development has changed significantly in the last 50 years. Even though processes, tools and best practices have changed, the importance of productivity measurement has not. Being able to assess productivity using an accurate metric is an essential part of managing a software team. Used correctly, the results of such measurements can be used to provide a baseline for the continuous improvement of teams and, by extension, the software they produce. In addition, productivity measurements are useful for predicting how software-under-development will evolve over time, an important aspect of managing the risks associated with software development.

For many years, productivity measurement was based solely on the amount of code produced during a given span of time. Though such a metric is easily automated, it has a numbers of drawbacks, chief amongst which are its inability to measure output other than program code and issues inherent to the differences between programming languages. Despite these shortcomings, LoC-based measurement is still in use today, perhaps due to its easily automatable nature.

Following the wide-scale adoption of Agile and Lean methodologies, detailed information on individual work units (tasks) in software projects has become easier to collect as many Agile and Lean methodologies mandate their use. Task data is often easily automatable and, in contrast to LoC-based measurement, suitable for measuring all types of development work. In essence, a completed task serves as a proxy for a given unit of output, be it source code, documentation or "invisible work" that produces no tangible artefacts.

Based on the aforementioned benefits, task data became the foundation for measuring velocity, arguably the most widely used indicator of productivity in the software industry today given the proliferation of Agile and Lean concepts. Though the metric has proved to be more flexible than LoC-based measurement, it stands in need of estimating the size and scope of each task. Using estimation as a part of a productivity metric yields results only as

accurate as the estimations themselves and is a time-consuming process.

In this thesis, it is assumed that even though tasks may vary in size and scope, all software projects in which the concept of a task is clearly defined have an implicit size interval within which all tasks lie, even without explicit estimation. Based on this assumption, we propose that, from a software developers point of view, simply counting the number of tasks completed within a period of time is a approach that is at least as valid as LoC-based measurement for general productivity measurement. Findings from the case study support this claim, lending credence to the theory that LoC-based productivity measurement can be replaced with implicitly estimated velocity in cases where a complete picture of productivity in a software development project is desired.

References

[Abr03] Abrahamsson, P.: *Extreme programming: first results from a controlled case study.* In *Euromicro Conference, 2003. Proceedings. 29th*, pages 259 – 266, sept. 2003.

[Alb79] Albrecht, Allan J: *Measuring application development productivity.* In *Proceedings of the Joint SHARE/GUIDE/IBM Application Development Symposium*, volume 10, pages 83–92. SHARE Inc. and GUIDE International Corp. Monterey, CA, 1979.

[BBM96] Basili, Victor R., Briand, Lionel C., and Melo, Walcélio L.: *How reuse influences productivity in object-oriented systems.* Commun. ACM, 39(10):104–116, October 1996.

[Bec99] Beck, K.: *Embracing change with extreme programming.* Computer, 32(10):70 –77, oct 1999, ISSN 0018-9162.

[BEEB98] Briand, Lionel C., El Emam, Khaled, and Bomarius, Frank: *COBRA: a hybrid method for software cost estimation, benchmarking, and risk assessment.* In *Proceedings of the 20th international conference on Software engineering*, ICSE '98, pages 390–399, Washington, DC, USA, 1998. IEEE Computer Society.

[Boe87] Boehm, B.: *Improving software productivity.* Computer, 20(9):43 –57, sep 1987, ISSN 0018-9162.

[Boe99] Boehm, B.: *Managing software productivity and reuse.* Computer, 32(9):111 –113, sep 1999, ISSN 0018-9162.

[Bro98] Brown, Charles E.: *Coefficient of variation.* In *Applied Multivariate Statistics in Geohydrology and Related Sciences*, pages 155–157. Springer Berlin Heidelberg, 1998.

[BSVW96] Blackburn, Joseph D., Scudder, Gary D., and Van Wassenhove, Luk N.: *Improving speed and productivity of software development:*

A global survey of software developers. IEEE Trans. Softw. Eng., 22(12):875–885, December 1996, ISSN 0098-5589. http://dx.doi.org/10.1109/32.553636.

[FP98] Fenton, Norman E and Pfleeger, Shari Lawrence: *Software metrics: a rigorous and practical approach*. PWS Publishing Co., 1998.

[FTB06] Fraternali, Piero, Tisi, Massimo, and Bongio, Aldo: *Automating function point analysis with model driven development*. In *Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research*, CASCON '06, New York, NY, USA, 2006. ACM.

[Giv08] Given, L.M.: *Qualitative Research Methods*. Number v. 2. SAGE Publications, Incorporated, 2008.

[HA99] Ho, V and Abran, A: *A framework for automatic function point counting from source code*. In *Proceedings of the International Workshop on Statistical Modelling*, IWSM '99, 1999.

[Hum95] Humphrey, W.S.: *A discipline for software engineering*. SEI series in software engineering. Addison-Wesley, 1995.

[iee98] *IEEE Standard for a Software Quality Metrics Methodology*. Technical report, December 1998, ISBN 1-55937-529-9.

[ifp13] *IFPUG Software Certification*. http://www.ifpug.org/?page_id=316, March 2013.

[IPF+11] Ikonen, M., Pirinen, E., Fagerholm, F., Kettunen, P., and Abrahamsson, P.: *On the impact of kanban on software project work: An empirical case study investigation*. In *Engineering of Complex Computer Systems (ICECCS), 2011 16th IEEE International Conference on*, pages 305 –314, April 2011.

[JRW01] Jeffery, R., Ruhe, M., and Wieczorek, I.: *Using public domain metrics to estimate software development effort*. In *Proceedings of*

the 7th International Symposium on Software Metrics, METRICS '01, Washington, DC, USA, 2001. IEEE Computer Society.

[KM86] Kirk, J. and Miller, M.L.: *Reliability and Validity in Qualitative Research.* Qualitative Research Methods. SAGE Publications, 1986.

[KMB04] Kaner, Cem, Member, Senior, and Bond, Walter P.: *Software engineering metrics: What do they measure and how do we know?* In *In METRICS 2004. IEEE CS.* Press, 2004.

[Koi13] Koivuluoma, Mika: *Cloud Software Factory.* `http://www.cloudsoftwareprogram.org/videos-and-webinars/webinars/i/27530/1500/tivit-webinar-mika-koivuluoma-cloud-software-factory`, March 2013.

[Lon04] Longstreet, David: *Function points analysis training course.* SoftwareMetrics. com, October, 2004.

[Mah04] Mahoney, Michael S.: *Finding a history for software engineering.* Annals of the History of Computing, IEEE, 26(1):8–19, 2004, ISSN 1058-6180.

[MAKS12] Münch, J., Armbrust, O., Kowalczyk, M., and Soto, M.: *Software Process Definition and Management.* Fraunhofer IESE series on software and systems engineering. Springer Berlin Heidelberg, 2012, ISBN 9783642242915.

[MCKS04] Mohagheghi, P., Conradi, R., Killi, O.M., and Schwarz, H.: *An empirical study of software reuse vs. defect-density and stability.* In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pages 282 – 291, may 2004.

[MF00] Maxwell, Katrina D. and Forselius, Pekka: *Benchmarking software-development productivity.* IEEE Software, 17(1):80–88, January 2000.

[MSBD04] McGhee, George R., Sheehan, Peter M., Bottjer, David J., and Droser, Mary L.: *Ecological ranking of phanerozoic biodiversity crises: ecological and taxonomic severities are decoupled.* Palaeogeography, Palaeoclimatology, Palaeoecology, 211(3–4):289 – 297, 2004.

[Pet11] Petersen, Kai: *Measuring and predicting software productivity: A systematic map and review.* Information and Software Technology, 53(4):317–343, April 2011.

[PP03] Poppendieck, M. and Poppendieck, T.: *Lean Software Development: An Agile Toolkit.* The Agile Software Development Series. Addison-Wesley, 2003, ISBN 9780321150783.

[RH09] Runeson, Per and Höst, Martin: *Guidelines for conducting and reporting case study research in software engineering.* Empirical Software Engineering, 14(2):131–164, April 2009.

[Rob02] Robson, C.: *Real World Research: A Resource for Social Scientists and Practitioner-Researchers.* Regional Surveys of the World Series. John Wiley & Sons, 2002, ISBN 9780631213055.

[Roy70] Royce, W. W.: *Managing the development of large software systems.* In *Proceedings of IEEE WESCON*, volume 26, pages 1–9, August 1970.

[SCC02] Shadish, William R., Cook, Thomas D., and Campbell, Donald Thomas: *Experimental and quasi-experimental designs for generalized causal inference.* Wadsworth Cengage learning, 2002.

[Sof13] Sof: *Software Factory Helsinki.* `http://www.softwarefactory.cc/`, April 2013.

[Ste46] Stevens, S.S.: *On the Theory of Scales of Measurement.* Bobbs-Merrill reprint series in the social sciences. Bobbs-Merrill, College Division, 1946.

[Sza07] Szalvay, Viktor: *Scrum Alliance — Glossary of Scrum Terms*. `http://www.scrumalliance.org/articles/39-glossary-of-scrum-terms/`, March 2007.

[Sza13] Szalvay, Viktor: *Scrum Alliance — Transforming the World of Work*. `http://scrumalliance.org/`, March 2013.

[TM09] Trendowicz, Adam and Münch, Jürgen: *Factors influencing software development productivity – state-of-the-art and industrial experiences*. Volume 77 of *Advances in Computers*, pages 185 – 241. Elsevier, 2009.

[VCW+84] Vosburgh, J., Curtis, B., Wolverton, R., Albert, B., Malec, H., Hoben, S., and Liu, Y.: *Productivity factors and programming environments*. In *Proceedings of the 7th international conference on Software engineering*, ICSE '84, pages 143–152, Piscataway, NJ, USA, 1984. IEEE Press.

[vV00] Vliet, H. van: *Software engineering: principles and practice*. John Wiley, 2000, ISBN 9780471975083.

[Wel12] Wells, Don: *Project Velocity*. `http://www.extremeprogramming.org/rules/velocity.html`, March 2012.

[Yin03] Yin, Robert K.: *Case study research: design and methods*. Sage Publications, 3rd edition, December 2003, ISBN 0761925538.

Appendices

A Collected productivity data in Project 1

Date	Team member A	Team member B	Team member C
03.09.12	0	0	0
04.09.12	0	0	0
05.09.12	0	0	0
06.09.12	0	0	0
07.09.12	0	0	0
10.09.12	0	0	0
11.09.12	0	0	0
12.09.12	0	0	0
13.09.12	0	0	0
14.09.12	0	0	0
17.09.12	1	0	0
18.09.12	0	0	1
19.09.12	1	0	1
20.09.12	0	1	0
21.09.12	0	0	0
24.09.12	3	0	2
25.09.12	2	2	1
26.09.12	4	1	3
27.09.12	4	1	2
28.09.12	0	0	0
01.10.12	2	0	1
02.10.12	1	6	2
03.10.12	0	1	0
04.10.12	0	0	0
05.10.12	0	0	0
08.10.12	4	5	4
09.10.12	1	1	1
10.10.12	4	4	1
11.10.12	0	0	0
12.10.12	0	0	0
15.10.12	5	0	3
16.10.12	0	2	0
17.10.12	0	0	0
18.10.12	0	0	0
19.10.12	0	0	0

Table 25: Programming tasks completed in Project 1.

Date	Team member A	Team member B	Team member C
03.09.12	0	0	0
04.09.12	0	0	0
05.09.12	4	2	2
06.09.12	8	7	7
07.09.12	0	0	0
10.09.12	3	4	3
11.09.12	2	3	3
12.09.12	2	1	2
13.09.12	3	2	2
14.09.12	0	0	0
17.09.12	4	3	1
18.09.12	1	1	2
19.09.12	3	1	2
20.09.12	3	4	3
21.09.12	0	0	0
24.09.12	9	0	2
25.09.12	3	2	1
26.09.12	5	1	4
27.09.12	5	2	3
28.09.12	0	0	0
01.10.12	5	5	4
02.10.12	5	9	3
03.10.12	3	2	1
04.10.12	0	0	0
05.10.12	0	0	0
08.10.12	6	5	5
09.10.12	4	3	2
10.10.12	8	4	2
11.10.12	0	0	0
12.10.12	0	0	0
15.10.12	5	1	4
16.10.12	0	2	0
17.10.12	0	0	0
18.10.12	2	2	2
19.10.12	0	0	0

Table 26: Total tasks completed in Project 1.

Date	Team member A	Team member B	Team member C
03.09.12	0	0	0
04.09.12	0	0	0
05.09.12	4	0	0
06.09.12	22	53	34
07.09.12	22	53	34
10.09.12	75	85	34
11.09.12	75	85	34
12.09.12	75	85	34
13.09.12	75	85	34
14.09.12	75	85	34
17.09.12	116	960	34
18.09.12	196	1009	56
19.09.12	196	2509	88
20.09.12	260	2509	88
21.09.12	2016	2509	88
24.09.12	2209	2509	110
25.09.12	2250	2640	502
26.09.12	2452	2670	612
27.09.12	2526	2670	614
28.09.12	2526	2846	614
01.10.12	2531	2940	650
02.10.12	2730	3049	934
03.10.12	2776	3049	1060
04.10.12	3020	3049	1060
05.10.12	3025	3049	1178
08.10.12	3132	3353	1251
09.10.12	3158	3522	1281
10.10.12	3456	3694	1397
11.10.12	3573	3694	1400
12.10.12	3574	3694	1400
15.10.12	3765	3831	1495
16.10.12	3838	3831	1495
17.10.12	3887	4560	1505
18.10.12	3887	4560	1505
19.10.12	3887	4560	1505

Table 27: LoC committed to repository in Project 1 (cumulative).

B Collected productivity data in Project 2

Date	Team member A	Team member B	Team member C	Team member D
29.10.12	0	0	0	0
30.10.12	0	0	0	0
31.10.12	0	0	0	0
01.11.12	0	0	0	0
02.11.12	0	0	0	0
05.11.12	0	0	0	0
06.11.12	0	0	0	0
07.11.12	0	0	2	0
08.11.12	0	0	1	0
09.11.12	0	1	0	0
12.11.12	1	0	1	0
13.11.12	1	0	1	0
14.11.12	0	0	1	0
15.11.12	0	2	0	4
16.11.12	0	0	0	0
19.11.12	1	1	0	0
20.11.12	0	0	1	0
21.11.12	2	2	2	1
22.11.12	2	0	2	0
23.11.12	0	1	0	1
26.11.12	2	0	0	0
27.11.12	1	0	0	1
28.11.12	2	0	1	0
29.11.12	1	0	1	1
30.11.12	0	0	0	0
03.12.12	0	0	0	0
04.12.12	0	1	1	0
05.12.12	0	0	0	0
06.12.12	0	0	0	0
07.12.12	0	0	0	0
10.12.12	0	0	0	0
11.12.12	0	0	0	0
12.12.12	3	1	1	0
13.12.12	0	0	0	0
14.12.12	0	0	0	0

Table 28: Programming tasks completed in Project 2.

Date	Team member A	Team member B	Team member C	Team member D
29.10.12	0	0	0	0
30.10.12	0	0	0	0
31.10.12	0	0	0	0
01.11.12	3	2	3	2
02.11.12	1	0	1	2
05.11.12	2	0	1	2
06.11.12	5	1	3	2
07.11.12	2	2	2	0
08.11.12	1	1	1	1
09.11.12	1	4	0	0
12.11.12	2	0	2	0
13.11.12	1	0	1	0
14.11.12	0	0	1	1
15.11.12	0	2	0	5
16.11.12	0	0	0	2
19.11.12	1	1	0	0
20.11.12	0	0	1	1
21.11.12	2	3	2	1
22.11.12	2	0	2	0
23.11.12	0	2	0	1
26.11.12	2	0	1	0
27.11.12	1	0	0	1
28.11.12	2	0	3	0
29.11.12	1	0	1	1
30.11.12	0	0	0	0
03.12.12	1	0	0	0
04.12.12	1	1	2	0
05.12.12	0	0	0	0
06.12.12	0	0	0	0
07.12.12	0	0	0	0
10.12.12	1	0	0	1
11.12.12	0	0	0	0
12.12.12	4	1	1	0
13.12.12	0	0	0	0
14.12.12	0	0	0	0

Table 29: Total tasks completed in Project 2.

Date	Team member A	Team member B	Team member C	Team member D
29.10.12	0	0	0	0
30.10.12	0	0	0	0
31.10.12	0	0	0	0
01.11.12	0	0	0	0
02.11.12	0	0	0	0
05.11.12	0	0	1	0
06.11.12	0	0	140	0
07.11.12	77	0	282	1646
08.11.12	126	0	479	2675
09.11.12	127	63	533	2766
12.11.12	127	63	540	2766
13.11.12	152	76	660	2830
14.11.12	355	181	927	2998
15.11.12	360	275	956	3061
16.11.12	388	280	1056	3061
19.11.12	433	440	1106	3061
20.11.12	705	566	1173	3412
21.11.12	815	567	1210	3597
22.11.12	890	567	1320	3728
23.11.12	987	567	1320	3728
26.11.12	1170	629	1320	3789
27.11.12	1225	629	1405	4067
28.11.12	1579	629	1489	4067
29.11.12	1625	887	1638	4147
30.11.12	1628	1055	1638	4483
03.12.12	1629	1118	1684	4483
04.12.12	1629	1118	1685	4623
05.12.12	1685	1158	1694	4644
06.12.12	1700	1262	1694	4646
07.12.12	1700	1262	1694	4646
10.12.12	1700	1262	1694	4646
11.12.12	1774	1288	1695	4648
12.12.12	1848	1288	1695	4648
13.12.12	1848	1288	1695	4648
14.12.12	1848	1288	1695	4648

Table 30: LoC committed to repository in Project 2 (cumulative).