Driving Software Quality and Structuring Work Through Test-Driven Development

Simo Mäkinen

Helsinki September 12, 2012

Master's Thesis

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

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

Tekijä — Författare — Author
Simo Mäkinen

Työn nimi — Arbetets titel — Title
Driving Software Quality and Structuring Work Through Test-Driven Development

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's Thesis	September 12, 2012	93 pages

Tiivistelmä — Referat — Abstract

Test-driven development is a software development method where programmers compose program code by first implementing a set of small-scale tests which help in the design of the system and in the verification of associated code sections. The reversed design and implementation process is unique: traditionally there is no attempt to verify program code that does not yet exist. Applying practices of test-driven design to a software development process—a generally complex activity involving distinct individuals working in an organization—might have an impact not only on the process itself but on the outcome of the process as well. In order to assess whether test-driven development has perceivable effects on elements of software development, a qualitative literature survey, based on empirical studies and experiments in the industry and academia, was performed.

The aggregated results extracted from the studies and experiments on eleven different internal and external process, product and resource quality attributes indicate that there are positive, neutral and negative effects. Empirical evidence from the industry implies that test-driven development has a positive, reducing, effect on the number of defects detected in a program. There is also a chance that the code products are smaller, simpler and less complex than equivalent code products implemented without test-driven practices. While additional research is needed, it would seem that the test-driven produced code is easier for the developers to maintain later, too; on average, maintenance duties took less time and the developers felt more comfortable with the code. The effects on product attributes of coupling and cohesion, which describe the relationships between program code components, are neutral. Increased quality occasionally results in better impressions of the product when the test-driven conform better to the end-user tests but there are times when end-users cannot discern the differences in quality between products made with different development methods. The small, unit-level, tests written by the developers increase the overall size of code products since most of the program code statements are covered by the tests if a test-driven process is followed. Writing tests takes time and the negative effects are associated with the effort required in the process. Industrial case studies see negative implications to productivity due to the extra effort but student experiments have not always been able to replicate similar results under controlled conditions.

ACM Computing Classification System (CCS):
K.6.3 [Management of Computing and Information Systems]: Software Management
D.2.9 [Software Engineering]: Management
D.2.8 [Software Engineering]: Metrics
D.2.4 [Software Engineering]: Software/Program Verification

Avainsanat — Nyckelord — Keywords
test-driven development, test-first programming, software quality, software testing, software process

Säilytyspaikka — Förvaringsställe — Where deposited
Kumpula Science Library, serial number C-

Muita tietoja — övriga uppgifter — Additional information

Contents

1 Introduction

Test-driven development [Bec03], often denoted TDD, is a software development method which drives developers to engineer program code incrementally through test code artifacts. According to the principles of test-driven design, sections of program code should always be preceded by associated test code if a test-driven development process is followed to the letter; the test code guides the developers in choosing the next programming tasks and verifies that a particular section of code is in agreement with the intentions of the developer. The reorganization of programming activities means that developers have to take new responsibilities [Bec00] and reshape their working patterns and existing ideas about software design in addition to other areas of software engineering.

A practitioner in software might consider the reasons behind such a transition. There is a chance that the reason lies in the issues identified in software engineering activities which have been acknowledged quite some time ago [Boe88]: primarily not being able to provide software solutions that would meet the demands of the customer without much wasted effort and be of acceptable internal and external quality. The lean philosophies that have originated from other industries address some of these concerns of quality on an abstract level [Ram98]. Industry experts, however, proclaim that test-driven development is a practice that concretely helps in providing quality products to customers [Cri06] and makes the developers feel better, more confident of their work [Bec03]; a praise for the restructuring of work and the tests created in the process.

The essential research question is that is there empirical evidence to support the claim of the industry experts? What kind of effects to internal and external quality attributes of products, processes and resources [FeP97] can be perceived when test-driven development principles are integrated into a software engineering process? Does software developed following these principles have better quality and if so, is it possible to show that the increased quality originates from practices of test-driven development and not from other factors?

Case studies and experiments can be used to explore phenomena in software engineering and given a controlled, stricter setting, also to explain the circumstances that lead to a specific outcome in greater detail [RuH09]. Therefore, empirical evidence from case studies and experiments can possibly explain whether test-driven development has an impact on software engineering activities and products.

Accordingly, in order to seek out answers to the research questions and to explore the phenomenon of test-driven development, a qualitative, unsystematic, literature survey on existing case studies and experiments in industrial and academic environments was carried out. Information on eleven distinct quality attributes was gathered from nineteen selected research reports which featured test-driven development. Quality was assessed by attributes which characterize the design quality of products and by other attributes which allow further analysis of the test-driven process and its effects.

The results of the review suggest that test-driven development has perceivable positive and negative effects on the quality attributes. Positive effects include a reduced amount of defects and reduced effort in maintaining a product after development; effects present especially in industrial environments. There were several cases where test-driven development seemed to help achieve simpler and better design quality of code both in industrial and academic environments. The effects that can be considered adverse have to do with the amount of extra effort developers need to put in to write test code which subsequently is reflected in the productivity of software development teams. The conclusion is that to some extent, the results support the claim of increased quality but quality is a multi-dimensional property and perhaps only preferences of individuals can decide the value of test-driven development.

This work is divided into six chapters. Following the introductory chapter, Chapter 2 sets the scene by discussing about the fundamentals of software engineering processes and describing through examples what activities are involved in software engineering; the chapter gives a frame to which test-driven development can be anchored and lines boundaries to the activities that can be affected in theory. Chapter 3 nails down the concept of software quality and covers certain measures that can be used to characterize quality in the form of quality attributes. Test-driven development and its directly related development methods are introduced briefly in Chapter 4. The qualitative literature survey and the review of the effects of test-driven development on individual quality attributes are covered in Chapter 5. Chapter 6 wraps up the information collected from the literature survey and concludes the work.

2 The Flow and Structure of Work in Software Engineering

The essence of software engineering is to produce software that resolves a specific, real world problem [Lam09]. All software is produced in a unique set of activities known as the software process and while unique, software processes share some common elements [Som11] which are illustrated in Figure 2.0.1. This chapter describes the key aspects of software processes.

Software specification entails acquiring knowledge about *why* the software is being developed, *what* services should the new system offer and *who* are involved in the operation of the system [Lam09]. Software specification is also known as requirements engineering [Som11] and it is central to the process as inadequate requirements will likely lead to incomplete software [PaG08]. Furthermore, catching requirement errors early is beneficial from a cost perspective as this minimizes the amount of lost effort [Lam09].

Software design and implementation builds upon the requirements that were elicited from the stakeholders or from existing systems, plans and documents [Som11]. The structure of software and its architecture define the kind of entities the development work is to produce; implementation details are then fixed in the programming activity.

Software verification and validation concerns both verifying that the implementation meets the specification and validating that the system being built fills the need of its users [PeY08]. Various analysis and testing activities can be performed on the development output or any other artifact that has been created.

Software evolution relates to the fact that software has to constantly adapt and shape itself according to changing needs [Som11]. Still, a certain life cycle of software development and software maintenance can be at times distinguished where development work is assigned differing amounts of effort.

2.1 Processes, People and the Environment

Software development takes place in rich environments and organizations with each organization possessing distinct cultural traits with sentient persons possessing varying skills and knowledge all operating as a part of some process to deliver software

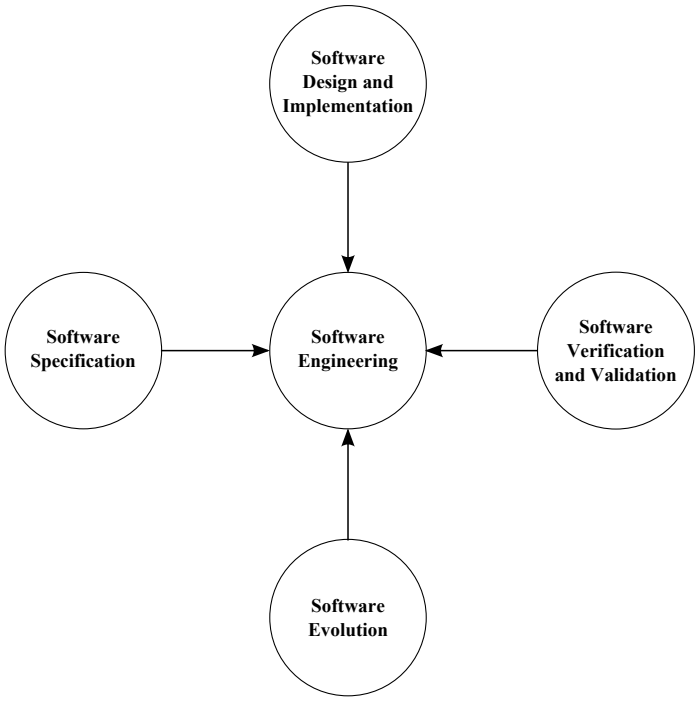


Figure 2.0.1: Software engineering processes share activities which address typical concerns in software engineering.

[Som11]. It is certainly not an easy task to define the best possible process for all organizations due to the nature of the organizations and differing application domains. Some processes and methods work better in other application domains than others.

Differences in process models arise more from the sequencing, length and nature of the activities [Som11]. The activities can be grouped into smaller units of work called *iterations* that can be repeated a number of times during the production cycle [ShW08]. This idea of grouping activities and performing multiple iterations is certainly not novel; Boehm presented the iterative spiral model already in 1988 and reported significant productivity gains over the older waterfall process models [Boe88]. Nevertheless, more recent process models emphasize iterative development and short iterations as well with iteration cycles of days instead of months or even years [ShW08, Kos08].

Agile Software Processes

True to their dexterous nature, these recent iterative and short cycle processes are commonly referred to as *agile software processes*. Boehm saw that one of the short-

comings of the waterfall model was that the document-driven approach and rigorous planning caused the development of program code that had no value to the stakeholders in the end [Boe88]. It is on these same grounds that the agile processes, a decade or so later, build on. In 2001, a group of software developers met together and wrote the *Manifesto for Agile Software Development* [BBB01] which outlines the principles of agile development:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan.

The authors behind the manifesto shared a worry that over the years, software development had taken a turn for the worse and become rigid as Highsmith narrates in his passage about the backgrounds of the manifesto [Hig01]. Technology and industrial processes in use at the time no longer served the interests of people and the well-being of employees and customers alike had become secondary. With the manifesto, the group wanted to reinstate humane values to software development with the hope of not only creating more pleasant working conditions but also to focus effort in work that would be valuable for the customer.

The term *agile* reflects a certain mindset according to the aforementioned principles rather than a strict formal process [ShW08]. The manifestations of agile ideas are included in such processes as *Extreme Programming* (XP) [Bec00] and *Scrum* which are amongst the most popular of agile software processes [ShW08, Kos08]. Extreme Programming, for example, is a full-fledged methodology how to conduct software development starting from how people are physically seated in a workplace and ending in the deployment of software [Bec00].

The Extreme Programming process model gives insight into all the distinct common software process tiers mentioned earlier and instructs on the practices of requirements elicitation, communication, design, programming and verification [Bec00]. *Test-driven development* is a tenet of Extreme Programming that turns the standard development activity upside down by reversing the order of validation and design [Bec03] but the practice is applicable as a *part* of other processes as well [ShW08, Kos08]. Test-driven development has a wide variety of implications and the practice is described in greater detail in Chapter 4.

Lean Software Development

In its quest for improvement, the software industry has over time sought best practices from other engineering disciplines as well to iron out issues that have been identified in software processes. Historically, these have been issues of cost and quality [Boe88] but records show that similar issues are still present [MFC05]. A source of inspiration for process refinement was offered by the Japanese automotive industry which had taken measures to ascertain that their output was of the highest of quality [Ram98]. These measures attracted the interest of the Massachusetts Institute of Technology that observed the methods and tried to understand the factors of success, leading to a publication in the early 1990's that coined the production technique as *lean production* [Ram98].

Lean production consists of elements of *value* which is a property to be defined by the customer, *value stream* relating to proper supply chain management and identifying unproductive activities, *flow* that stresses the need of fluid transitions between activities, *pull* which is a concept of minimizing inventory through only-on-order deliveries and *perfection* as an idea of always striving for better outcome in all activities with continuous improvement [Ram98]. Encompassed in these elements is the key concern of lean production which is the avoidance of *waste* or *muda* in Japanese, referring to anything that is not of value in the process. Furthermore, quality in lean production can be seen as the responsibility of all participants; quality is not only determined by external observation which can be seen as waste to the process but rather better judged by those who are involved in the production of the output at the same time keeping in mind that it is the customer who defines value and the requirements for quality.

There has been some debate whether the lean methods can be adapted by the software industry since the methods were originally designed for the automotive industry [Ram98]. The aircraft and defense industry in the United States, which both rely heavily on software, at least took initiative in the 1990's to find out how to incorporate the lean principles. Application of lean production methods has also been done in more traditional software companies.

A seasoned software company in the United States that was in the business of creating software related to construction had been operating for 30 years when it realized in 2001 that its software processes had become costly and far from optimal [MFC05]. With lean principles in mind, over the course of two years the company made significant improvements and was able to improve productivity, scheduling and quality as

for example the time it took to maintain program code during production went down by as much 60% or over. Another study of several software development companies showed that lean production methods were helpful in increasing communication within the companies [TLS07].

The objective of lean production is to avoid unproductive activities, streamline work and focus on the customer, objectives which are not that far from those of agile processes. Perhaps it could be argued that agile software processes as such are already partly implementations of lean production methods. The examples do show that there's always room for improvement in software processes and bringing lean production methods to the table might just provide the common framework needed to get started with process refinements [MFC05].

The Sociotechnical Environment

Processes structure work and various process models offer generic guidelines for how and in which order to conduct activities but the social dimensions of organizations should also be noted. Organizations themselves are path dependent; past actions have an effect to choices made [CaW11]. Moreover, organizations consist of social individuals with personal capabilities, relations and other contextual properties that affect the bond between the individual and the organization and give motivation.

Working in an organization is a group effort and collaboration shapes the way people behave and solve problems [CaW11]. Thus, organizational structures can direct the flow of work with different structures having different collaboration patterns [RoJ10]. *Mechanistic* structures have a higher degree of work specialization and centralization which might not satisfy employees in creative professions whereas *organic* structures are less formal, almost boundaryless and can be more satisfying to work in for some people [RoJ10].

Organizations can consist of different types of work groups and teams, some of which can be relatively independent self-managed work teams particularly characteristic of organic structures and some can be *virtual* in nature [RoJ10]. Virtual teams are distributed teams whose members are not necessarily physically in close proximity but are connected through the use of computers and communication networks. Virtual team members don't enjoy the same social environment as those members who share the same physical space; this can hinder communication and result in reduced job satisfaction [RoJ10, CaW11]. In fact, sharing a workspace can have an impact

not just on communication but generally on team productivity as well [TCK02].

Teasley et al. conducted a study at an automobile company where several small software development teams of under ten people were put to work together in open workspace rooms for about four months [TCK02]. Due to the close quarters, communication improved as people were for example able to casually overhear the conversations of other team members and the pilot project teams clearly outperformed previous software development teams with productivity numbers two times greater than dispersed teams. By previous theory, to increase communication the team must be seated within a certain close distance of each other and it is simply not enough for the team to inhabit scattered areas of the workplace for the effect. For these reasons, collocated teams and open workspaces are encouraged as a part of certain agile process models, like Extreme Programming [Bec00].

Although virtual teams and organizations have a collaborative handicap, means exist to alleviate the shortcomings by constructing virtual communities where the members can freely exchange their views [CaW11]. For instance, the Mozilla Foundation is an international organization that takes advantage of the volunteer based open source community where hundreds of people from different countries can join in a virtual organization to develop a specific software project so that Mozilla only has some coordination responsibility [CaW11]. Mozilla uses various forms of asynchronous communication methods in its projects such as mailing lists, forums and on-line information spaces to allow participants to commonly work on project tasks and get feedback on ideas; this can give a sense of participation to project members but the asynchronous method can also be discouraging for minorities if some ideas are not acknowledged at all. Virtual organizations have the option to use synchronous communication methods like text chats as well though this requires simultaneous presence of members that might not always be possible.

Overall, the environment in which software engineering takes place can be described as a complex *sociotechnical system* comprising of interconnected entities such as machinery, software, organization, people and the supporting society [Som11]. Software development needs to acknowledge the existence of these tiers to understand the role software plays in the whole system and as a part of other sociotechnical systems.

2.2 Identifying Requirements

Requirements engineering aims to specify the objectives of the system under development and the entities that are involved in reaching those objectives [Lam09]. The entities in the environment can be hardware, software or people who can be labeled as stakeholders to signify those individuals or groups who contribute to the future system in some way. Requirements can be broken down into *functional requirements* which define the set of functional services the system should have and *non-functional requirements* that define more how these services should be produced and what limitations are in place for the system.

The process of requirements engineering has several distinct but interleaved phases: *requirements elicitation* which focuses on requirements discovery and understanding the field of the system, *requirements specification* where requirements are documented in some formal or informal manner and *requirements validation* that verifies a match between actual requirements and those specified in documents to avoid unnecessary rework [Lam09, Som11].

Stakeholders are a mixed group of people in different roles that have varying aspects and concerns about the future system; this information can be translated into requirements for the system [Som11]. Because of the diversity of the people involved, the views of different stakeholders can be conflicting and knowledge transfer can be hindered by poor communication channels and expertise gaps between stakeholders [Lam09]. Involving the right stakeholders yields more correct and complete requirements but given an environment, there is no specific way to identify these stakeholders [PaG08].

It is possible to attempt to single out stakeholders by evaluating the predominance and the centrality of people in the social context that can either be an internal organization, an external organization involved in the development of the system or a completely external party such as a demographic user group [PaG08]. There are some agile software process models such as Extreme Programming that rely on a smaller sample of only several stakeholders per project that are trusted to represent the communities they belong to but there the selected stakeholders are given a more substantial role in the software development process requiring a daily presence with the software development team [Bec00].

In practice, a recent survey report of several hundred software development organizations from China showed that communication about requirements is mostly done

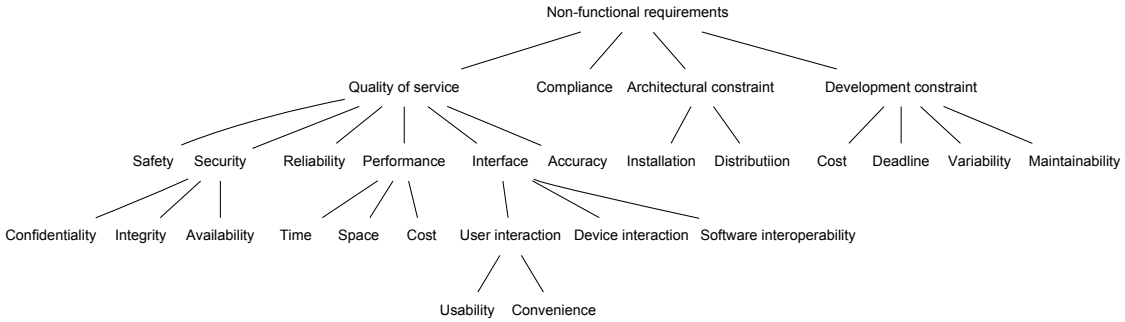


Figure 2.2.1: Non-functional requirements place constraints on the whole system [Lam09].

with management layers and less with actual system operators [LLP10]. While the surveyed organizations included a good number of offshore departments for non-domestic organizations it is possible that the results reflect cultural properties to some extent; Asian countries have in the past scored high on power distance metrics when analyzing cultural diversity meaning hierarchical manager-subordinate relations might be more strict than in countries that have a lower power distance [JaM10]. Cultural differences shouldn't be overemphasized and generalizations about large populations should be avoided [JaM10] but this could theoretically have an effect on horizontal communication and on the reported manager-centric requirements elicitation.

Requirements Elicitation

Requirements can be obtained from suitable stakeholders by using *stakeholder driven* techniques that are based on different interaction methods between the software developers and other stakeholders or *artifact driven* elicitation techniques which also involve stakeholders but can focus on documents and other static sources to gain understanding about the environment, actors and existing systems [Lam09]. Artifact-driven techniques are particularly purposeful in attaining some non-functional requirements such as performance; non-functional requirements are constraints and quality requirements for the system as depicted in Figure 2.2.1 and this type of information might not be available directly from stakeholders.

Interviews are examples of commonly used stakeholder driven elicitation methods where stakeholders are consulted for their knowledge in structured and closed or unstructured and open interviews [Lam09, Som11]. Another form of interaction is narratives that take advantage of stories or scenarios created together with the

stakeholders to describe a requirement or a specific path of events in the system.

While scenarios can depict operation of the system under normal operating circumstances with positive scenarios and abnormal conditions with negative scenarios, it might not be trivial to capture the intended behavior of the system in all cases with them due to the great amount of possible scenarios [Lam09]. Not all the stories look the same, either; Extreme Programming uses lighter, short stories which are then augmented with additional information from the stakeholders which is why heavy stakeholder involvement and presence during development is essential to Extreme Programming [Bec00]. Requirements can also be acquired through indirect measures as sharing tacit knowledge can be more effective by illustration so stakeholder observation in their natural environment by ethnographic or other study is a possible method in requirements elicitation [Lam09, Som11].

Requirements Specification

Discovered requirements for the system should be recorded for further use as specifications in some informal manner such as natural language descriptions complemented with visual aids or in a more formal and structured manner that is less ambiguous but which might be harder to understand [Lam09, Som11]. Formal specifications benefit from predefined structures allowing automated interpretation [Lam09]. Scenarios can for example be written in a semi-formal language so that they can then be used to validate the system and its behavior [CAD10]. The Chinese requirements engineering survey reports the prevalence of informal specification methods; the majority of about two out of three preferred diagrams of various sorts and less than one out of ten were using formal specification methods [LLP10]. Rather few, about a fifth, reported to use natural language documents while stories were somewhat more popular but not as widely used as diagrams.

Requirements can be grouped in a requirements document listing both the functional and non-functional requirements for future reference [Lam09]. A well structured and comprehensive requirements document can support the software engineering process, especially for critical software systems where safety is a key concern [Som11]. Although, the need for the requirements document has recently been contested: Extreme Programming for example questions such a need and replaces structured requirements documents with tighter collaboration with the stakeholders [Bec00]. Avoiding unnecessary documentation is one of the basic principles of agile software processes and Extreme Programming reflects this principle.

Requirements Validation

Requirements that have been specified and documented might be incomplete in a number of ways: the specifications might miss out important requirements, the requirements could be conflicting, ambiguous or specified in such a manner that makes the evaluation of the requirement impossible [Lam09]. Discovering such deficiencies in requirements at a late stage can incur cost due to the rework needed to make the system fit the real needs of stakeholders [Som11]. As an example, a software development organization in China was engaged in development of a three month project, only to find at the very end that the stakeholder actually had a different view about the requirements which effectively doubled the development time [LLP10].

Requirements review is one of the most prominent requirements validation methods where stakeholders inspect the documented requirements in the hope of identifying flaws in requirements [Lam09]. Formal validation methods are available as well, given that the specifications have been written in a formal language [Lam09]. System prototypes for stakeholders can be used for validating requirements, too [Som11]. Simulating a model and providing visualization through animation is akin to prototyping but is more model centric [Lam09].

Agile practices and Extreme Programming see requirements validation more as a continuous process [Bec00]. Constant flow of information between involved stakeholders, on-demand requirements and frequent, almost weekly iteration demos offer the stakeholders the chance to validate the requirements [Bec00]. Heavy stakeholder involvement is also present in validation methods which base on acceptance tests [Kos08]. Acceptance tests are tests partly prepared by the stakeholders for whom the system is being developed so that the stakeholders themselves specify the criteria for success validating their own requirements.

2.3 Designing and Developing Software

Turning requirements into a working system happens through stages of design and implementation, where a general plan known as the *software architecture* is first drafted for the particular piece of software followed by implementation i.e. the act of programming [Gar00]. Architecture in software has not always been regarded that significant and prior to 1980's, different software architecture styles had little coverage outside the drawing boards of developers and organizations where the software had been developed in [ShC06]. Development of software requires creativ-

ity and individuals have different working habits on the personal level but there's evidence that a good understanding of the personal process and the application of sound engineering principles can lead to a better outcome [Hum95].

Software Architecture

Software architectures and the plans of such architectures are concerned with the interrelations of the major software components of the system, providing an abstraction which shows the responsibilities of the major components [Gar00]. Software architectures help to understand the system before and after construction; the architecture itself can also be analyzed and evaluated to assess how well the system satisfies quality attributes [Gar00, ShC06]. Quality attributes, addressed further in Chapter 3, can help to characterize the quality of software from the standpoints of those involved in the development, management or operation of the system in some way.

It has been shown that software architecture has a significant effect on the system's different quality attributes [Som11] and architectures designed with care have been tied to success of entire projects [Gar00]. In addition, architecture controls the aspect of reuse relating to the ability to provide a reusable architectural pattern which other systems can benefit from. Evolution of the software system and the possibilities of extension are directed by the architecture as well and by this property the maintenance of the system is affected [Gar00].

Architecture of software needn't always be designed beforehand to have a solid software structure as *incremental design* techniques can be applied [ShW08]. Part of the agile mindset, incremental design allows constant change to the architecture, built little by little and changing as required. According to this viewpoint, greater insight about the whole system can be gained over time which leads to inevitable architectural modifications and thus software architecture shouldn't be considered as something rigid and unchanging but flexible.

Incremental design is not only limited to architecture but it can be practiced on a smaller scale, too [ShW08]. Software development in an object-oriented environment involves the design and modeling of objects that translate into programming entities known as *classes* which have *operations* or *methods* that can alter the state of these objects. Classes can reflect the properties of real, physical world objects in such detail that is necessary to the software system. Objects and classes are not available

in all programming environments, though. Nevertheless, incremental design can be performed on classes and methods; this is one of the key concepts of test-driven development [ShW08], described further in Chapter 4.

Pair Programming

Programming solo is one approach to implementing a design, yet there exists an alternative method of implementation called *pair programming* which is integral to Extreme Programming [Bec00]. In pair programming, the same physical workstation is shared by two individuals who collaborate and work on the same piece of program code, exchanging views and trying to maintain a state of flow where a good deal of attention can be paid to the task at hand. Roles are divided so that one person is the *driver* whose responsibility is to handle the input devices and program; the other person is the *navigator* who can think ahead and suggest the driver how to proceed with implementation at the same time having an eye on the work of the driver. To keep up the drive, the roles are switched frequently and people are rotated from one pair to another, even daily.

The premise is that as a pair, the individuals are more effective and can work faster, delivering higher quality output than as two individuals alone. Wray identifies four different mechanisms of how pair programmers benefit from the close collaboration [Wra10]. By the first mechanism, programmers are able to solve problems efficiently because they're able to verbally express their thoughts and concerns to the other partner. Apparently the formulation of the thought as spoken words complemented with a suitably insightful question from the other half of the duo is enough to nudge the brain in the right direction towards solving the problem. The second mechanism is based on the fact that a second pair of eyes offers a fresh aspect to the program code and issues are less likely to go unnoticed. Programmers who work in pairs are also not so easily inclined to take shortcuts in programming and can thus achieve better quality than individuals who are at times allured by the easy way out: this is Wray's third suggested mechanism. Finally, according to the fourth mechanism the work of the whole team becomes more predictable as a result of pair programming because team members are more aware of the skills possessed by others in the team.

Programmers do at least *think* they're better off programming in pairs. Begel and Nagappan performed a survey at Microsoft where about a hundred experienced programmers were asked how they felt about pair programming [BeN08]. The majority of programmers thought that pair programming allowed them to create program

code that had fewer defects and many saw knowledge sharing as a benefit. About a half believed that working with a pair would result in better overall quality of program code. While the programmers saw benefits in pair programming, a substantial three out of four found that such kind of programming is inefficient in terms of using resources as the two people have to invest time in the same activity with the additional burden that pair programming is less flexible with individual preferences for working hours.

It seems that the programmers are not far off from the right track there. Arisholm et al. investigated the effects of pair programming by carrying out an experiment with around 300 professional programmers from some of the Nordic countries and the United Kingdom [AGD07]. One third of the programmers were tagged as individual programmers whose performance was compared to the rest of the group working as pairs. Programmers with little experience did fewer mistakes particularly in complex tasks when paired up but the more seasoned professionals did a better job as individuals than in pairs. While in some cases pairs were slightly faster, it took almost the same time for the pairs to complete the tasks as for individuals. The worry of the programmers at Microsoft is reflected also in this result: summing up the overall time expended by both members of the pairs, even in the best case the required effort was almost one and a half times greater than that of the individuals and over two times greater in the worst case. It is to be noted that programmers that took part in the experiment didn't have much previous experience on pair programming and the experiments lasted only for a day for any given pair or individual, both factors which might downplay the effects of pair programming [AGD07].

In an academic setting, pair programming has proven to be of some help to students and passing computer science courses can be more probable as a result of pair programming practices [BEW08]. This finding looks like to be in line with the study of Arisholm et al. that stated junior programmers to benefit most from pair programming, at least related to the amount of errors [AGD07]. But for a professional environment where programmers are more experienced, it is harder to justify pair programming as a method of implementation because of the extra effort needed to get the job done.

2.4 Verifying and Validating Software

Software verification and validation are two distinct activities that aim to establish the relative level of quality in software in a given environment [PeY08]. Together

these activities form the concept of *verification and validation* (often abbreviated V & V) where verification tries to ensure that requirements for the software are being met and validation tries to make sure that the requirements are meaningful to the operation of the whole system in the first place.

Software testing is incorporated in software verification; it is something which can be performed on any product of the software process that has a deterministic output relating to a specific input [MVM10]. The target of software testing should be an executable artifact such as program code but also formal specifications and models can be tested. Since the need for execution, software testing can be considered to be a set of dynamic verification and validation techniques.

Static verification and validation techniques are those where execution is not needed [MVM10]. For example, requirements validation through review sessions is a validation technique that doesn't require execution of the artifact but just a group of stakeholders gathered for the review. Similar static inspections can be carried out on program code by software developers in order to catch errors [Som11]. The recommended team size for program code inspection is about three to four people and such a team can be more effective in defect detection than dynamically executed tests; inspections can in some circumstances uncover over two times more defects.

It is to be noted that software testing and the entire field of verification are at an inherent disadvantage. Due to the very basic nature of computation, it is impossible for a machine on its own to verify the correctness of a program [Sip06]. This classic problem is called the *halting problem* and it refers to the issue that a machine cannot decide what kind of results are acceptable and what aren't.

At the same time, only a limited number of inputs can ever be verified because the interdependency of different program inputs creates conditions where the amount of input combinations quickly reaches astronomical heights [MVM10]. Hence, by definition, software verification is an incomplete activity; the implication is that verification cannot assure the absence of defects [Dij72].

Software Testing

Software testing starts with planning what needs to be tested and specifying the expected results in some manner. After the tests have been specified, they can be executed and the outcome can be recorded for further study to see which part of the program didn't comply with the test specifications made earlier [MVM10].

More precisely, a test specification is commonly referred to as a *test case* which associates a well-defined input with expected behavior [MVM10]. When the test case is executed under a given set of conditions, the preconditions, the actual result should match the value defined earlier. If there is a discrepancy between the actual and the expected result, the test case is said to *fail*. A collection of individual test cases forms a *test suite*.

Test cases should be selected with care, not least due to the computational complexity involved. Suitable test cases can be identified using various criteria; test cases that help to achieve a greater *coverage* in terms of structure or some other property can be preferred, for instance [MVM10]. Coverage is a term used to measure how completely the test cases are able to invoke statements in the executable artifact, most often the program code. A high structural coverage of a program means that more program statements are being executed by test cases, but a full coverage can hardly be attained and in some cases a structural coverage of about 85% is considered a good objective [Kos08]. Besides structural coverage, coverage of other stated objectives can be the criteria as long as these objectives are verifiable. For example, a list indicating the most prominent aspects of the program can be used to narrow down the test cases and serve as coverage criteria [MVM10].

To further reduce the amount of test cases, the necessity of test cases can be evaluated. For instance, a minimal set of test cases can be uncovered by comparing the overlap of executed statements by test cases, i.e. determining if there are multiple test cases that cover the same part of program code and thus be redundant [MVM10]. Additionally, different weights can be assigned to test cases giving more importance to specific test cases over the others [MVM10].

When it comes to selecting test cases, experience is a fine tutor, too. Practice has shown that certain input categories are more prone to cause problems with software so values from these categories should be used in test cases [PeY08]. Dividing input dimensions to categories is called *partitioning* where the idea is to cluster input values with common characteristics and then select samples from within these clusters. Especially input values close to the boundaries of the clusters or partitions are of interest. Expert knowledge helps to realize potentially problematic scenarios as well [Som11] and this knowledge can be used to construct fault models on which test cases can be based on [MVM10].

Software testing is carried out by different stakeholders during different phases of the software process [Som11]. *Development testing* is performed by developers or other

stakeholders involved in the development phase and can initially focus on individual, partly incomplete components while gradually expanding to testing larger program entities and their interrelations. *Release testing* looks at the system as a whole and testing on relatively complete system components can be conducted by third parties that might not be part of the development team in order to evaluate whether the system or some part of it is ready for release. *User testing* takes place closest to end users of the system; these users are able to give feedback about the operation of the system in a setting that resembles actual operating conditions. User testing can take the form of acceptance testing where the system is tested according to the acceptance criteria set by the key stakeholder, the customer [MVM10].

The division of testing responsibilities is not always clear-cut. Release testing stresses the need of an external testing team, and this division of testing responsibilities can be seen in certain development teams that separate the role of a tester and a developer as well [Som11]. Extreme Programming stresses that the responsibility shouldn't be shifted to people who are perceived to be outside of the development team but quality should be the concern of everyone [Bec00]. Extreme Programming teams rely on *automated tests* that have been constructed by quality conscious developers and *manual tests* are left for more exploratory testing purposes.

In exploratory testing, developers and testers can rather freely together or separately manually inspect the software and use personal insight to create and execute custom test cases on the fly, uncovering such potential errors in the software that might have been missed by the set of automated tests [KBP02]. Automated tests that are executed by a machine are efficient in verifying the consistency of software especially when the software has undergone changes at a later stage in development but automated tests cannot replace manual testing of some non-functional properties that require a pair of trained eyes for evaluation, such as properties of aesthetic and visual nature [Som11].

Automated tests come in various shapes and sizes [ShW08]. The smallest automated tests are lightweight tests known as *unit tests* which target program units of methods or classes and are intended to run as fast as possible. For example, in Extreme Programming, most of the automated tests that are created should be unit tests [Bec00].

Testing of interrelated entities that aims to verify the compatibility of the individual components with each other is called *integration testing* [PeY08]. Integration tests focus on the communication between components in more depth and are thus slower

to execute [ShW08]. This *interclass* testing can involve as many classes as required by a particular interaction scenario which can be derived from the dependencies of the classes with the hope of revealing faults undiscovered by unit tests [PeY08]. Both unit and integration tests can be categorized as development tests, as these are normally made during development by those with programming skills [Som11].

Unit test frameworks that can be used to specify test cases, execute tests and record test results are available for many different programming languages [Kos08]. For example, *JUnit* is a popular unit test framework for the Java programming language [BeG98].

Listing 2.4.1 shows an example of a test case, annotated by the test identifier in a Java class that contains tests. In this example, the scenario is first set up and a method is executed from the class under test; the method `studentCount` of the class `University` is the *unit* in this case. Based on the input that has been defined for the scenario, there is a specific output that is expected as the result after executing the method. The result is *asserted* by comparing the actual result with the expected result and the assertion holds if the values are the same. When all assertions of a test case hold, the particular test case passes; the test case fails when an assertion doesn't hold or there is an error in the test code. Whether failing or passing, the unit test framework can save the result of the test case which can be referenced later on to see which units and test cases didn't conform to test specifications.

Isolated and automated tests are fundamental to test-driven development and unit tests are good for this purpose [Bec03]. Isolation of unit tests is two fold in object-oriented software: intraclass testing isolates testing to a specific class at a time [PeY08] while isolation of individual test cases ensures that each test can be executed independent of each other [Kos08].

Code Smells

Beyond software testing with explicit test cases, there exists more fuzzy measures that can be used to locate segments in program code likely to be the cause of errors in software [FBB99]. Code smells are such indicators; telltale signs of bad programming practices which by expert opinion are known for causing trouble. A couple of dozen bad practices were originally identified and dubbed as bad code smells. Some are tied to non-functional properties that have an effect on the longer term maintainability of the software product but some of the practices can more or

Listing 2.4.1: A Unit Test Case in Java

```
public class UniversityTest {

  @Test
  public void testStudentCount() {

    University uni = new University();

    Student maryStudent = new Student("Mary");
    Student johnStudent = new Student("John");

    uni.addStudent(maryStudent);
    uni.addStudent(johnStudent);

    int expectedStudentCount = 2;
    int actualStudentCount = uni.studentCount();

    assertEquals(expectedStudentCount, actualStudentCount);
  }

}
```

less directly lead to or hint of errors in software.

According to Fowler et al., the various bad practices have a different impact on the qualities of software and of all the bad practices, program code duplication can be seen as one of the major contributors to problems in software [FBB99]. Zhang et al. studied this relation between faults and duplication of code in addition to several other bad smells that could be observed from the structure of the program code in two open source projects [ZBW11]. The evidence suggests that code duplication is indeed connected with an increase in faults as code segments which had repeated, similar code patterns were more prone to errors than segments that didn't repeat the same structure.

Additionally, it was found that parts of code that had long message chains contained a relatively higher proportion of faults than parts without long message chains [ZBW11]. Message chains are hierarchical structures of code that need to be traversed to a certain depth in order to obtain some piece of information required by the particular code segment, which creates a dependency between the different nodes in the chain and makes the program more vulnerable to errors when modifications

are made at some point [FBB99].

Bad programming practices and poor program code structures seem to eventually lead to faults, but not all practices are as bad as they seem since not all code smells are tied to increases in faults [ZBW11]. Some patterns of bad programming practices can be detected automatically which indicates that these fuzzy verification techniques could be used without manual inspection of program code; better code structures would in the end translate to software with fewer defects.

2.5 Software Evolution and Embracing Change

Software, akin to living and organic entities, has a cycle of life. Although the life span of software is not constant and is dependent on environmental factors, software systems need to stand the test of time even for decades [Som11]. During this time, it is likely that there is need for the software to solve real world problems it was not initially designed to solve. The software needs to adapt and change: it needs to *evolve*.

Active phases of software development can be scattered along the entire life of software but after some time, software development activities can be gradually discontinued if the software is no longer fit for purpose [Som11]. At some point in its development, the software is considered mature enough to be handed over to the customer; the software is effectively released. This begins the *software maintenance* phase where faults can be removed from the software system and the system can be changed either due to environmental, technical changes or according to the feedback received from users and other stakeholders. Over time, maintenance duties can be carried out by different groups of software developers who might not be as familiar with the system as the original developers, which might make maintenance more difficult.

Software shows the signs of old age, too [FBB99]. Repairs to software in the maintenance phase might be done in haste without considering alternatives which might be more suitable for the software system in the long run and the structure of software deteriorates as time passes. Bad structures can be replaced later with the help of *refactoring*, a practice that aims to improve the design of software. Even major architectural modifications are possible but these carry a greater cost than simpler improvements such as removing code duplication or splitting up methods that have become too long. Agile practices recognize the need for incremental refactoring; the

design of software should be evaluated constantly during development and refactored accordingly to rid the software of flaws in design made earlier [ShW08].

Lehman's Laws of Software Evolution

Over the course of several decades, Lehman studied software systems and summarized his observations of the software processes as *Lehman's laws* [Leh96]. One source of data for the research was IBM and the system development performed at the company in the turn of the 1970's. Lehman admits that the word *law* here might be too strongly put as the circumstances of software development are hardly fixed. Nevertheless, the following eight laws can be used to characterize software evolution.

1. Continuing Change
2. Increasing Complexity
3. Self Regulation
4. Conservation of Organisational Stability
5. Conservation of Familiarity
6. Continuing Growth
7. Declining Quality
8. Feedback System

Lehman's first law presents an idea why software systems need to evolve and change. The change must be continuous because the operating conditions are in constant state of flux which creates the need for the software system to change; a process which can only be controlled through appropriate feedback mechanisms. But the change comes at a price: each modification made to the software system carries the risk of making it more complex and future maintenance efforts more laboursome. This fact is recognized in the second law.

Laws three and four relate much to the organizational context of the software development body. According to Lehman's third law, organizational behavior governs the evolution of software as organizational structures dictate who is interested in

the end product. People in different positions will apply different criteria of success possibly unique to the organization in question. There are limits to the control of the organization, though. The fourth law states that there is considerable outside influence that affect the organization's software development activities for a given system and this tends to make the flow of modifications relatively constant.

As software grows in size, its structure will get more complex and individuals will have to spend more time to acquaintance themselves with the software code in order to perform development tasks. Lehman's fifth law is linked to the fact that development effort is partly affected by how familiar individuals and groups are with the system. There might be ways to preserve that familiarity as Gîrba et al. shows how code ownership can be traced from a version control system even for large software system containing thousands of source code files [GKS05]. Perhaps the ownership of a particular source code file could be used when assigning work for individuals so that those who are the most familiar with a particular source code segment would be given tasks related to the segment.

The sixth law states that when users of a system become more accustomed to using the system, there will be continuous demand for new functionalities and the improvement of existing ones; it is not only the environment that drives the change but the growing familiarity of the users. Incorporating the new functionalities will lead to a system bigger in terms of size. Over time, there can be a shift in user expectations and the system is expected to be capable of performing tasks it was not originally designed to perform. The bar is set higher and although the system can be operating without failures, the quality of the system is seen to gradually diminish in the eyes of the users. By the seventh law, this kind of reduction of quality is unavoidable if feedback of users is disregarded but in the end it is possible to maintain perceived quality given that action is taken to handle the feedback.

Finally, it is the eighth law that stresses the importance of the feedback system. Higher levels of development activity are so associated with higher levels of feedback. In essence, feedback drives development and maintenance in an organization.

Instruments of Software Evolution

The capacity of software to evolve can be described in finer detail by decomposing evolvability to elements that can be measured to some extent as well [CHH01]. Highly evolvable software should be *analyzable*, *changeable*, *stable*, *testable* and *com-*

pliant with good maintainability practices.

A software product is analyzable if it takes only a short time to pinpoint the section that would need to be modified according to new requirements or for other modification purposes such as correcting the behavior of existing code. The product is changeable if it doesn't take too long to make the modification which could be measured by the relation between effort and the size of the modification. To be stable, the system should function as well or better after the made modifications so the amount of observed failures shouldn't increase due to the changes. Unstable systems exhibit unpredictable effects throughout the system after changes. When a product is testable, re-verification of the changed system is rapid. Finally, compliant products follow relevant conventions and standards like coding conventions.

Following coding conventions is by expert opinion relevant to the maintenance and development of software although experts have varying views which conventions should be followed more closely or not at all [SGH11]. Software code can be automatically checked with style checkers that analyze code and report coding violations. Smit et al. performed a study of several large open source projects that at best had several hundred thousands lines of code and analyzed the violations. For instance, a recent version of the Apache Ant codebase had a high number of multiple string literals and magic numbers that are considered bad programming practice and problematic for maintenance. Missing code comments were also prevalent in all of the analyzed projects.

3 Defining and Measuring Quality in Software

What does it mean when software is considered to be of high quality? Software quality can be said to comprise of meeting previously recorded requirements but also more broadly fulfilling the needs of users and customers [Gal04]. The definition of quality may include tacit assumptions such as that the software is developed in a professional manner and is relatively free of defects. This chapter describes how different dimensions of software quality can be defined and what kind of measures can be used with software.

3.1 Models of Quality

Elements of quality can be grouped in *quality models* that try to address a multitude of possible quality concerns. One of the early attempts of creating such a quality model was by McCall and fellows who created a report about software quality factors for the U.S. Air Force back in 1977 [MRW77]. The report identified and categorized quality factors to a temporal dimension where different factors are important in the stages of *operation*, *revision* and *transition* of the software product. For instance, the operation stage has factors of correctness of the product as in how well it fills the needs of the users, reliability, usability, efficiency and integrity. In the product revision stage there are for example factors of maintainability and testability which have a close match to the elements of evolvable software depicted decades later. Finally, product transition has factors of portability, usability and interoperability. While factors of cost and time were not part of the actual quality model, the report does mention these two to be valuable when evaluating quality factors.

McCall's work has certainly not been in vain. The quality model has many characteristics that are useful in assessing the quality of software and McCall's tree model remains well referred in modern day literature [Gal04, FeP97]. The factors in the model can be compared to the functional and non-functional requirements of software. Correctness corresponds to the functional requirements of a system and fits well to the predominant theme of software quality in so that primarily software should do what the user expects of it. Besides correctness, other factors in the model are more associated with non-functional requirements which are considered to be somewhat equal to quality requirements of a system [Lam09]. Indeed, the resemblance between McCall's model and Lamsweerde's graph of non-functional requirements as depicted in Figure 2.2.1 is striking.

3.2 From Errors to Failures

Developing software is an error prone business, mistakes happen. Problems can have different labels depending at what stage the problems are detected [FeP97]. The first thing that occurs is a human *error* in typing in code or in understanding requirements, thus creating a product of work that is incorrect. When the error goes unnoticed it turns into a *fault* that is then present in the code or other artifact which can cause subsequent errors and faults. After some time during the operation or testing of the system, it might be noted that the system is not performing as

expected and so the discovered fault becomes a *failure*. By chance, it is possible that some faults never lead to failures if the particular code segment containing the fault is not executed during normal operation of the system. The term *defect* is a collective term that usually contains faults and failures while the term *bug* is used with faults.

The reason why errors exist is that humans are fallible. Galin lists nine causes of software errors which all originate from human behavior [Gal04]. Communication is seen as one important theme; requirements might not be specified in an adequate way and the cause of error might be *faulty requirements definition* or for some reason the message is not going through due to *client-developer communication failures*. It might also be the case that the developer simply ignores the requirements so the error could be said to stem from *deliberate deviations from software requirements*. Even if requirements are properly understood and communication with the customer is fluent, *documentation errors* might still originate from documentation that is flawed in some manner with such recorded statements that contribute to false assumptions about the product, its environment or any other associated product that is used in the development of the system.

Design and implementation might be problematic too with occasional *logical design errors* and *coding errors* of differing complexity. If documentation or instructions are not followed as they should be, the cause for error could be *non-compliance with documentation and coding instructions*.

Processes can fail as well; inadequate testing or other *shortcomings of the testing process* lead to errors not being noticed in time. Furthermore, mistakes can happen when developers and other users are operating a user interface of sorts. When there's deviation from the correct sequence of actions and the incorrect action sequence ends up causing an error, it can be considered as a *user interface procedure error*.

3.3 Measures in Software

Measurement in software can be performed on a number of objects that have a different role in software development. These objects that contribute to the development of the software product can be *processes* that consist of the activities used in practice, *products* such as program code or other artifacts like documents about the product and *resources* in an organization consisting of personnel and also the physical office environment where the activities take place [FeP97].

Objects that belong to these categories have attributes that can be considered to be either *internal attributes* or *external attributes*. The difference between internal and external attributes is that internal attributes are more straightforward to measure by inspecting the object in question while external attributes are more associated with the environment of the object and are harder to measure having less objective meaning by themselves. For instance size is an internal attribute of a product or a resource that can be measured with relative ease and usability is an external attribute that is more difficult to measure as the attribute is dependent on the person using the system and the surroundings where the system is operated. Table 3.3.1 lists a number of internal and external attributes that can be used with processes, products and resources.

In addition, there are attributes which can be said to be both internal and external such as complexity; a complexity value can be directly calculated for program code but the complexity of various designs can also be evaluated. Because of the different nature of the attributes, the measurement scales involved can vary. Internal attributes that are directly observable can for example use ratio scales to state that the complexity of a program code artifact is 13 but for the complexity of the design a nominal scale ranging from not very complex to very complex might have to be used if the observation is based on human evaluation. Of course, the same holds for any attribute be it either internal or external: a proper measurement scale must be used for the attribute.

Besides internal and external attributes there are *direct* and *indirect* measures which are distinct from the former two [FeP97]. Direct measures are measures that can be readily measured like the internal attributes. Size of a software product is an example of such a measure. Indirect measures on the other hand are formed by combining many attributes together and calculating the result based on some relational formula. Here, defect density can offer an example of an indirect measure: it can be calculated from the number of defects and the size of the product. In a similar fashion, an indirect measure for programmer productivity can be formed from the produced product size measured in lines of source code and the effort used to produce this output.

It should be noted that despite the existence of such an indirect measure, the concept of productivity in software development has a different meaning than in traditional industrial environments [FeP97]. Effective effort is difficult to measure as the software development process is a unique process and no two are quite alike. Measuring

Attribute	Attribute Type
Size	Internal
Time	Internal
Effort	Internal
Price	Internal
Number of defects found	Internal
Test coverage	Internal
Complexity	Internal and external
Usability	External
Maintainability	External
Reliability	External
Productivity	External

Table 3.3.1: Measurable objects have directly observable internal and environment specific external attributes.

size ignores other quality aspects and using a size attribute such as lines of code doesn't take into account the functional components completed. It is suggested that a measure known as function points would be used instead of source lines of code to have a more meaningful indirect measure [FeP97].

Glass made a remark about programmer productivity by taking a look at historical research made on the subject and illustrated the differences in productivity [Gla03]. A study made in 1968 reported some programmers to be 28 times more productive than others in the study while more conservative opinions from the 1970's state that some programmers can be 5 times more productive. Glass thus argues that it is people, not processes which matter the most and emphasis should be put on personal skill when looking for new talent to hire. Glass implies in this statement that measuring productivity has a meaning as these differences exist but he also stresses that it is not easy to come up with suitable experiments and measures for finding good people. In practice, there has only been a weak correlation between such experiments and actual work performance.

3.4 Metrics for Characterizing Product Qualities

Out of processes, products and resources, products still tend to be the centerpieces of measurement. Gòmez et al. reviewed existing literature about software measures and found out that almost eighty percent of related articles deal with measuring software products instead of process or project measures [GOP08]. In these articles, there were a number of attributes that appeared to enjoy a higher degree of utilization than others. By far, measures of complexity and size had the highest popularity followed by class inheritance and defect related attributes. Structural and time attributes also had their share of references but attributes such as cohesion and coupling were already less referred to.

How much do these different attributes and metrics then speak of the quality of software? It would seem that at least some of the metrics are associated with perceived quality [BWD00]. Briand et al. made a study with students on several projects that each lasted for some months and extracted a large set of metric data from the code classes. Finally, in order to obtain defect data, the products were put under the scrutiny of testing professionals. In this setting with somewhat inexperienced students, it was noticed that those code classes which had a higher level of coupling also contained more defects. A similar relation was not found with a number of other metrics; for instance cohesion metrics were not tied to increase in defects.

Code metrics can be good indicators of quality but they are not the best. Misirli et al. compared the defect prediction capabilities of more traditional static code metrics with graph based network metrics and change based *churn* metrics [MCM11]. Code churn is a measure that can be measured by comparing different versions of source code files and evaluating the amount of change in these files [NaB05] whereas network metrics can be obtained from dependency graphs of components which indicate the relative importance of individual components in a connected network [ZiN08]. Nagappan and Ball originally concluded that a defect prediction model using churn measures like modified, added and deleted source code lines was effective in predicting where the defects would be [NaB05]. The Microsoft server system that Nagappan and Ball studied had some 44 million lines of code (sic) but Misirli et. al confirmed these kinds of metrics to work for somewhat smaller systems as well.

In the study of Misirli et al., three different defect prediction models were built based on code, network and churn measures, respectively. The prediction models were fitted to data from a system that had been under development for 20 years and the defects were categorized according to their detection time of functional,

system or field testing. Upon closer inspection, churn measures bested code measures containing metrics for complexity and structure of code classes in all defect categories. Network metrics were also no match for churn metrics. Overall, churn metrics seem to predict defect density with a high level of confidence; errors are introduced specifically to areas which are being modified the most.

When code metrics *are* used, it is reasonable to ask when is the quality of software considered good or bad with these metrics if the metrics can indeed be used to determine the quality? It is by no means straightforward to define where to put the limits but Ferreira et al. try to suggest boundaries that can be used for this purpose [FBB12]. Their approach was to take a selected set of 40 different open source software products and analyze object-oriented metrics from the source code of the products. Ranges of values which appeared most often were labeled as good, those not that often as regular and the rare ones as bad. Overall, the open source products were found to be low in coupling, high in cohesion and have shallow inheritance trees.

The analysis of Ferreira et al. suggests that quality thresholds for static code metrics can be set. For instance, the inheritance depth of classes should be no more than two, classes shouldn't have more than one coupling and the amount of public fields should be limited close to zero. To assert that the limits can be used in identifying problematic classes, Ferreira and her group zoomed in on a number of classes that were classified as bad regarding a selected set of metrics. The code classes classified as bad by the metrics seemed to have structural and other problems when inspected manually, i.e. they were of lower perceived quality as other classes in general. Although the study covers only a few metrics for a single programming language, it does indicate that static code metrics could at least be used in detecting specific sub-standard software product areas.

Size

Measuring the size of a software product involves looking at the *length*, *functionality* or *complexity* of the product [FeP97]. Besides methods of counting, there are some other ways size can be measured by evaluating the information content of a program and comparing the similarity of program components but these methods require more complex calculations [AGG07].

While length of other products like documentation could be taken into account when measuring size, usually it is the length of program code which is considered

in counting the length [FeP97]. The length metric typically used for program code is called *lines of code* (LOC) and the metric is measured by counting how many source code lines there are in the source code files; blank lines and code comments are often omitted. However, different coding styles and formatting of source code can produce dissimilar readings for similar program code files and there is some disagreement what actually counts as a line of code. Also, not all program code is written by hand as code can be produced by automatic means or copied from other sources.

Because of the ambiguities of counting source code lines, it has been suggested that size could be measured by functionality and the calculation of *function points* is one approach towards this direction [FeP97]. Function points try to capture the amount of features in a software product through the amount of data sources the system has to process and produce. The data sources can be inputs of various sorts resulting from interactions with the user, outputs generated by the system or program files required internally. Processing of all resources is not of equal effort, so different complexity weights can be assigned to the processing of these resources. Function points are language independent but not always accurate and the complexity weight assignations are quite subjective which mean that function points have limited applicability.

McCabe's Cyclomatic Complexity

The flow of program code can be illustrated using *control flow graphs* that are segmented by code methods containing individual source code statements [PeY08]. Source code statements in such diagrams can be grouped in *basic blocks* which contain statements next to each other that do not branch in any way. For instance, statements that modify the local state of variables can be inside one basic block but a new basic block is inserted to the graph if there is a condition that can branch the code in different directions. McCabe suggested in the 1970s that the amount of basic blocks and their relations in control flow graphs could be used to derive the complexity of a program component [McC76].

The theory behind McCabe's complexity measure is that the sheer size of a program component is an inadequate indicator of complexity but the amount of distinct paths tells an entirely different story about the component. Programs have better structure when there are not an excessive number of these paths and this should result in components which are easier to test and understand.

McCabe's cyclomatic complexity can be calculated with the following formula where e is the number of edges in the graph, n is the number of nodes in the graph and p is the number of connected components. It can be considered that most control flow graphs have only one connected component so normally the value of p is fixed to one.

$$v(G) = e - n + 2p$$

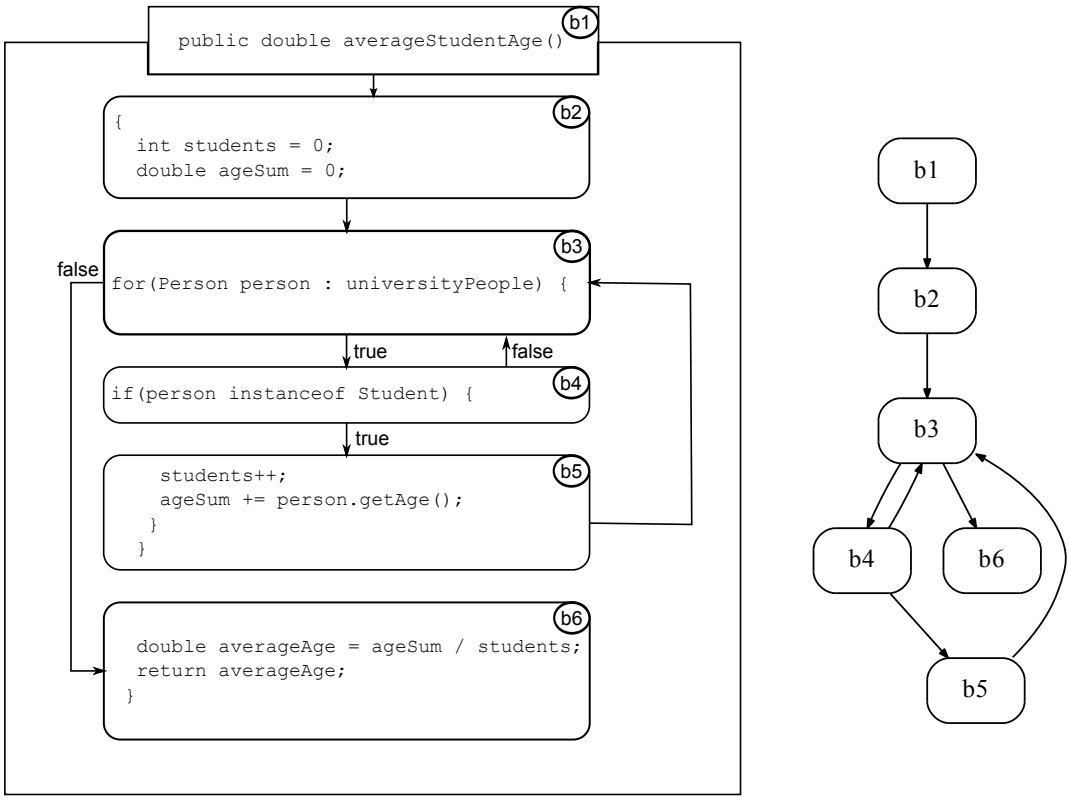
In the Fortran era of the 1970's, McCabe concluded that well structured programs generally had a cyclomatic complexity value under 10. In abnormal cases, the complexity ratings were reported to go beyond the 50 mark and subsequently these programs were also seen less reliable than programs which were not as complex.

Serving as an example, Figure 3.4.1 illustrates a section of Java code that has been divided into sequential basic blocks. Considering the amount of basic block nodes and the edges between the nodes, it is possible to derive the cyclomatic complexity of the Java method. Given the illustrated example, there are seven edges and six nodes yielding the assignment and result $v(G) = 7 - 6 + 2 = 3$. So the complexity of the program is three, which according to McCabe means a well structured program; although originally developed for languages like Fortran, the notion of complexity itself should also hold for Java programs.

Coupling, Cohesion and Other Object Oriented Metrics

Programs that have been written in an object-oriented programming language offer the chance to evaluate the relationships or the relative integrity of the objects included in the design. Chidamber and Kemerer suggest six different metrics that can be used to analyze the structural consistency and connectivity of objects [ChK94]. The theoretical background in their study is backed up by formal analysis and an empirical evaluation of two distinct applications written in C++ and Smalltalk with a total of some 2000 classes. It is argued that with the help of these metrics, a level of architectural control could be attained and deviating components could be pointed out, promoting reuse at the same time.

The first of the six metrics, *Weighted Methods Per Class* (WMC) is essentially the sum of complexities of each method in a class. Here, the premise is that when classes grow in complexity and the number of methods increases, they will be increasingly more difficult to maintain and reuse of the classes is hindered. Chidamber



(a) A control flow graph of a Java method **(b)** A simplified control graph

Figure 3.4.1: A control flow graph can be constructed from source code statements (a) which can be reduced to a control graph containing only the basic blocks (b) for determining McCabe's cyclomatic complexity for the program.

and Kemerer intentionally leave the definition of complexity open in this case and theoretically any suitable complexity measure could be used when computing the metric. In the sample projects, not many classes scored over 10 in this metric but there were several central classes where the value was over 100; the reusability of such specialized and complex classes can be questioned. Although the complexity measure used to calculate the metric for the sample projects wasn't specified, the classes which had the highest metric scores subsequently had the highest amount of methods.

The layout of the classes is another matter worthy of attention according to Chidamber and Kemerer. Object-oriented programming languages make inheritance of other classes possible which leads to reuse of properties from higher levels of the class hierarchy. *Depth of Inheritance Tree* (DIT) measures the distance to the root of the tree from the class, i.e. how many classes are in the inheritance hierarchy be-

tween the first class and the class in question. Deep inheritance trees are not always desirable as this increases complexity of the design but then again on the positive side classes deep in the hierarchy can benefit from reuse. For instance, the sample set contained a class which had over 130 methods, most inherited from its eight ancestor classes. It was noted that such high depth can affect testability. Generally, the inheritance trees were seen to be quite flat and the majority of classes only had a few ancestor classes.

Besides looking at the depth, the amount of immediate descendants of any class can be inspected by the *Number of Children* (NOC). The classes that end up having many children are quite central and can be considered targets for more rigorous testing. However, having too many children is not advisable as it can hint that the class has too many roles and further division of responsibilities could be possible. Most classes evaluated in the study had no children at all and thus were relatively independent but there were a few classes which had around 40 to 50 subclasses, playing a more central role in the design.

When classes execute methods or access the fields of other classes, the classes are said to be coupled. *Coupling Between Object Classes* (CBO) is a metric intended to capture this behavior; it is a count of the amount of couplings the class has. If a class has multiple connections to other classes, it follows that it becomes less independent and vulnerable to ripple effects from the coupled classes; changes in functionality elsewhere may be passed on to the class itself. The loss of independence is accompanied by a loss in reuse as using a highly coupled class in another context is strenuous. Dependence on other classes can be stronger or weaker based on the amount and type of data passed between the classes [FeP97] and there are coupling metrics which distinguish the type and direction of coupling [FBB12]. While the distinction can provide additional information of class associations, originally the definition of CBO didn't specify such a distinction. Similar to the other described metrics, Chidamber and Kemerer stress the need of additional testing for highly coupled classes.

The metric evaluation showed that coupling might be programming language specific as the Smalltalk application had significantly higher metric values than the C++ application. Typically, classes weren't associated with other classes at all but many Smalltalk classes had nine connections or over whereas with C++ very few had over zero. On the other end of the spectrum, a class with over 200 connections was identified in the Smalltalk application.

Coupling deals with the amount of interactions a class has to other classes and so does the *Response for a Class* (RFC) metric. Instead of counting the set of classes, the response set used to evaluate the metric drills down to the method level in a more specific way. Every method invoked from inside methods of a class slightly builds up the complexity as the more methods are involved, the harder it becomes to track the flow of the program. Thus, the response set is the set of distinct methods summed up from each method of the class and the response metric is the plain size of the set i.e. how many different methods can be invoked in total. The difference between programming languages shows in the size of the response set: half of the Smalltalk classes invoked 29 methods or more with maximum metric values soaring beyond 400 units and for C++ the values were some four times lower.

The last of the six metrics presented by Chidamber and Kemerer indicates how consistent a class is internally judged by the use of shared instance variables of methods. This metric is known as *Lack of Cohesion in Methods* (LCOM) and it is derived from the count of methods which don't use any common instance variable of the class minus the count of methods that do share the use of a common instance variable. The theory is that if different methods don't use the same instance variables, the abstraction of the class is less than perfect. Classes where the methods are unrelated could benefit from spreading the functionalities into a number of other classes to keep each class as cohesive as possible. There was not much evidence of the lack of cohesion in the sample data and most classes scored close to zero but one case was found where the lack of cohesion was 200; a truly multi-functional class with many roles.

Not all have been convinced of the proposed object oriented metrics, though, and there has been some controversy over the theoretical validity especially regarding the coupling and cohesion metrics. Hitz and Montazeri argued that the definition of coupling fails to consider indirect method invocations appropriately and that the specification of cohesion is partly inconsistent [HiM96]. It is shown that there are situations where the lack of cohesion should increase when a new method is added to a class but by the effective definition cohesion doesn't change. An example of sequential methods that share common instance variables also partly refutes the idea behind the lack of cohesion when the amount of methods increases over a critical threshold in the scenario. Based on these findings, Hitz and Montazeri propose a new cohesion model which takes the identified weaknesses into account.

4 The Test-Driven Paradigm

It sounds unintuitive at first, to verify and validate something that doesn't exist. But yet this is the core idea of test-driven design, to write down tests before proceeding to a more detailed implementation of the section of program code in question [Bec03]. This paradigm shift in software development inverts the flow of work of design, implementation, validation and verification activities but further developments of the concept concern other development activities as well.

This chapter covers the fundamentals of test-driven development practices. The essential ideological foundation of the development practices is test-driven development that employs developer-written automated tests to drive organic design of software applications [Bec03]. Acceptance test-driven development builds on a similar idea but the focus there is on tests that are prepared in close collaboration with the customer [Kos08]. Behavior-driven development employs higher-level scenarios and stories to serve the same ends as acceptance tests but is not limited merely to the higher level since the practice also instructs how development on lower levels should proceed [CAD10].

4.1 Test-Driven Development

Test-driven development is about taking small building blocks of software and constructing a larger software entity in the process. As described by Beck, an advocate of the development practice, the construction of software in such manner should be accompanied with adequate ways to verify that the constructed block of program code is fit for use [Bec03]. The practice revolves around automated testing and specifically the order in which the tests are created relative to program code.

Prior to writing a test, there is usually no existing implementation of functional code that is to be executed by the test; the test comes first of all. Precisely, the inversion of design and implementation makes it possible to have incremental design patterns where the design and implementation of tests precedes the design and implementation of program code. The automated tests written are unit tests for specific methods of classes but other product elements such as view components of web applications can be driven test first, too [Kos08]. Besides being drivers for design, the tests are imperative in stating whether the product is being built the right way. Because of their central role, writing tests should be the responsibility of developers which will shift much of the burden of verification from non-developers

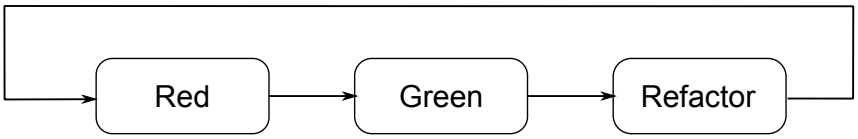


Figure 4.1.1: The development cycle of test-driven development advances rapidly from failing to passing tests and ends up in polishing the implementation.

and transform the nature of testing [Bec03].

A key factor in test-driven development is speed. Tests should be fast to write and they should be fast to execute. The first step in the development cycle is to rapidly create a test that will fail due to missing implementation i.e. a test that will produce a *red* color in a test execution environment. Initially, the test will reflect a tiny design idea that will direct the coding activity for the following moments by first stating the desired outcome. Next, the task is to create a method or a whole class that instantly leads to a working *green* state with minimal effort and implementation. After reaching a state where the test executes without failure, the implementation is to be cleaned up or *refactored* to remove some of the shortcuts that were taken in the hasty process from red to green. Refactoring is essential to the design of the program lest the code would suffer from duplication and other non-generic structures. Figure 4.1.1 depicts the development cycle known also as test-code-refactor.

The hypothesis is that such incremental design and repeating the cycle over and over again results in programs that have better structure. Beck argues that the design of the program is sounder because excess code that doesn't correspond to any test will be cut off leaving out unnecessary functionality and the design elements are smaller as well. It is mentioned that having plenty of tests that can be executed frequently might help the developers cope with change and create a more pleasant working atmosphere where the developers have greater confidence in their work.

Choosing which kind of tests to use for driving the design has its challenges. Writing code only against a failing test supposedly increases the structural code coverage of tests as writing them after the implementation would be a task not easily undertaken. Nevertheless, developers have to choose the scenario they're using: working with scenarios from the *happy path* that are void of exceptional and erroneous conditions is one way to start writing tests [Kos08]. Preparing a minimal list with some keywords of tests to write might help in selecting which test to write next and such a list can be of use when thinking about the abnormal scenarios as well.

To enable rapid advances from failing to passing tests, Beck suggests three strategies. With the idea in mind of minimizing the time of having failing tests, *fake it until you make it* refers to writing impure code that is solely written to make a specific test pass; perfection can wait until the green bar. *Triangulation* on the other hand means writing several tests from slightly different angles for the same part of code which will guide the developer towards the right solution. When the implementation is trivial enough, following the pattern of *obvious implementation* leads to direct implementation of functionality based on the test.

4.2 Acceptance Test-Driven Development

Test-driven development gained a foothold along with Extreme Programming in software engineering in the turn of the millennium but practitioners noticed that something was amiss with the process. Stories prepared for the iterations outlined the requirements of the customer and yet reviews at the end of the short iterations showed that the program developed by the programmers didn't match the needs of the customer [Rep04]. Thus, there was a demand for a mechanism that would allow developers to be more aware of the intentions of the customer.

Watt and Leigh-Fellows suggested that the Extreme Programming process could be improved by augmenting the process with additional steps to confirm that the acceptance tests prepared together with the customer would have all the necessary information [WaL04]. Bringing acceptance tests to the limelight, the approach was first called acceptance test-driven planning. The same line of thinking continued with the methods of storytest-driven development (STDD) [Rep04] and customer test-driven development (CTDD) [Cri05] which both emphasized that acceptance tests should be the starting point for development and that the criteria in the acceptance tests should be illustrated with concrete, explicit, examples prepared together with the customer. Eventually—when the concept matured—the ideas from these methods converged into acceptance test-driven development (ATDD) which closely follows the footsteps of its predecessors.

Koskela writes that acceptance-test driven development is a methodology where user stories from the customer are converted into acceptance tests which can be executed and eventually implemented by programming [Kos08]. The advantage is that the due to the stronger link between the customer and the developer, the customer can verify which stories are of interest and specify the conditions which indicate a successful implementation of the scenario. At the same time, acceptance tests serve

as a valuable indicator of progress.

According to Koskela, development driven by acceptance tests has four stages. The first step is to *pick a story* that has been defined by the customer. Stories that correspond to a requirement of the program can be written in a specific format and language but, most of all, the customer should be comfortable with it. A number of stories might have been prepared in advance so the selection of the next story to process implies its priority.

After a story has been selected, *writing tests* out of the story should be the next activity. Again, working with the customer, a rough sketch of the acceptance tests should be made that illustrates what actions are involved with the story. Once the desired actions have been decided, more detail should be added to the scenarios stating the set up and the expected outcome.

When there's an adequate understanding about the scenario and the acceptance test has been written down, there's need to specify the test in a manner that is executable by a machine i.e. to *automate the test*. For instance, the translation to an executable test can be done using table-like structures where each row stands for an action in the scenario that has a determined result. Frameworks like the Framework for Integrated Tests (Fit) [MuC05] allow such translations and theoretically the tables could be filled by the customer that has ownership of the acceptance test. However, programmers are required for the final conversion and connecting the test scenario to program test code.

The final step is to *implement the functionality* specified in the acceptance test that is red, in the failing stage. Compared to driving with unit tests, acceptance tests are coarse grained and they take much longer to implement. The implementation of acceptance tests doesn't have to but can take advantage of test-driven development; implementing one acceptance test can take multiple standard test-driven cycles. The whole acceptance test cycle is complete when the acceptance test ultimately passes as defined in the scenario.

4.3 Behavior-Driven Development

As a spin-off from the original test-driven concept, behavior-driven development (BDD) extends the idea by including behavioral descriptions of features that contain essential requirements of related stakeholders; this practice resembles the definition of acceptance tests together with the customer. Chelimsky et. al write that the

behavior driven concept is more specific than acceptance test-driven development as it describes in greater detail what kind of elements to use for the construction of scenarios and in which stage feature definitions should be done in the development process [CAD10].

The ideology behind the development practice is that when the implementations matches the higher-level scenario well, there is minimal waste in the development process as the customer has specifically expressed the need for such a feature and accepted the scenarios that represent various outcomes. It can be seen useful to describe the features and scenarios in the language of the domain in question to properly reflect the terms and phenomena and to enhance communication with the customer.

Chelimsky et al. describe how customer requirements are broken down into different components in behavior-driven development [CAD10]. *Features* are the functional requirements of a customer in the behavioral design pattern. A feature can contain multiple *stories* that outline the interrelations of stakeholders, features and benefits. Stories can be laid out by a generic format such as Connextra [NoD03] which uses the template terms *as a*, *I want* and *so that* to respectively specify the role, feature and the business value for a story. A story can have many *scenarios* that are examples how certain situations play out. There can be positive scenarios that describe successful interactions and negative scenarios that describe more exceptional conditions. Each scenario can have many *steps* which are the smallest behavioral indicators.

Chelimsky et al. mention that there is a language called Gherkin which defines a grammar to use with scenarios [CAD10]: it uses the keywords such as *given* for the scenario set up, *when* for actions, *then* for results, *and* and *but*. Gherkin has been translated to a number of natural languages which facilitates communication between the customer and the developers and there are a number of programs that can interpret descriptions based on the language. Overall, the scenarios work as acceptance tests that are approved by the customer thus giving direction to the development of a particular piece of software. Finally, detailed implementation of the scenarios can take advantage of unit tests and test-driven development to complete the cycle.

Listing 4.3.1 serves as an example of Gherkin. The story is first described in the Connextra format indicating the rationale for the scenario which in this case is the lecturer's interest to communicate with students. A positive, happy path, scenario

Listing 4.3.1: A sample scenario illustrating the Gherkin language

```
Feature: Lecturer cancels class
  As a lecturer
  I want to inform students of a cancelled class via an SMS
  So that I can serve the information needs of students accordingly

Scenario: Cancellation sent succesfully
  Given I am lecturing a course "Test-Driven Development"
  And I have an enrolled student "Mary"
  And the student has the phone number "+358-40-1234567"
  When I select the next class of the course
  And I click on "Send cancellation"
  Then a short message should be automatically composed
  And the message should have the name of the course
  And the message should indicate the word "cancelled"
  And the message should end with the cancellation date
  And the message should be sent to the enrolled students
  And I should see the message "Message sent successfully"
```

with multiple steps follows the story definition which state the objective that the developer should try to reach given a set of initial conditions. Such an explicit scenario communicates the requirements of the customer to the developer and the format of the story encourages to think of the value that is created for the customer upon the completion of the story.

5 Benchmarking Test-Driven Development

It has been established that there are various ways to assess the quality of software products and the processes surrounding the development of software. Developers *feel* that test-driven development is better: Beck mentions that it gives developers more courage and reduces fear [Bec03] while Koskela writes that it's a pathway to greater confidence and reduced stress levels [Kos08]. Crispin is on the same track and mentions that from the perspective of an experienced tester, projects using test-driven development practices seemed more successful [Cri06].

The question is that to what extent does test-driven development actually help developers and is it possible to observe the effects of test-driven design by looking at the changes in quality characteristics? This chapter tries to answer the question and

evaluate the impacts to both the internal and external attributes of products, processes and resources with the help of results recorded earlier from previous studies.

The review focuses on eleven distinct quality attributes which are covered in the following order. An analysis on *defects* allows to evaluate the defect reduction capabilities of test-driven development while *coverage* indicates the verification strength of the tests written in the process. The internal design quality of products is analyzed by attributes of *complexity*, *coupling* and *cohesion* whereas *size* of code products is used to assess whether there's a tendency for the products to grow or shrink when applying test-driven principles. Design and development time taken in the whole process by individuals and teams is characterized by *effort*. The end-user, customer, viewpoint is shown in the evaluation of *external quality*; an evaluation which includes qualitative information about test-driven development from developers. The external attribute of *maintainability* shows how changeable a product is and how at ease the developers are when performing maintenance duties. Finally, the output rate of teams and individuals is characterized by *productivity* which leads to a brief consideration of *cost*.

While seeking for answers to the research question, there needs to be an understanding what to use as the yardstick when comparing test-driven development and other forms of development. The other forms of development used as references are some development practices where testing takes place after development and design of code is not done incrementally through tests; several studies use the notion of *test-first* for test-driven development and *test-last* for the more traditional method of design, implementation and verification.

Concerning the validity of results, it is good to bear in mind the validity threats introduced in the various studies covered here. A substantial amount of the studies have been conducted in academic contexts such as universities using undergraduate or graduate students in their experiments which might introduce a selection bias and pose a threat of internal validity to some of the studies. In many cases, the studies do acknowledge the validity threats but it needs to be noted that the results might not apply as such in all environments where the test subjects are more mature as developers.

5.1 Previous Work

There have been surveys in the past which have examined a number of studies about test-driven development and tried to infer its effectiveness or the lack of it in specific areas of software development. Shull et al. reported on an aggregate study of the kind a few years ago which looked at over twenty different articles and the individual studies contained in those articles [SMT10].

The survey categorized studies based on their experiment type and assessed whether the arrangements of the experiments met the requirements of a well defined study; more weight was given to experiments that were of the highest quality. Each study that was selected to the survey was tagged so that those showing a positive effect with test-driven development on some dimension of quality got a label indicating that they were better regarding that dimension and if the results were worse or there was no significant difference between the development practices, the studies were labeled accordingly.

Many of the studies mentioned defects and the impact the development practice has on the number and density of defects. Mostly, it was found that test-driven development was better in this respect, especially in an industrial context. However, there were also studies that reported no differences and in a minority of the cases the results were actually worse.

Theoretically, incremental design patterns could have an effect on the structural properties of code such as code complexity, size and coupling. According to the studies, this is not necessarily the case as improvements to structural properties were only observed in half of them. In fact, the other half showed structural product quality to either decrease or not change at all and so Shull et al. conclude that the evidence is inconclusive in respect to internal quality factors.

A test-driven development strategy focuses much effort on writing tests and the extra work could be seen as a factor of productivity. A clear pattern wasn't visible in the studies and the results on productivity differed between study types. Experiments performed in an industry setting quite consistently showed productivity to drop but more controlled experiments suggested just the opposite.

The survey reported by Shull et al. bears resemblance to the literature survey that is covered in this chapter but the research method is slightly different. While the researchers in the former survey grouped internal and external attributes to several distinctive categories, the survey performed here drills down to individual quality

attributes in order to study the effects of test-driven development more closely. Here, the analysis of the existing research on test-driven development is mostly qualitative.

5.2 On the Effects on Internal Attributes

Internal attributes speak volumes about the artifacts created in the software process, the individuals participating in the process and the process itself. Structural attributes such as complexity, size, code coverage and coupling can be extracted directly from program code. The effort spent on a project is another variable that can be measured without much difficulty. Counting the defects from a defect database or document is equally effortless although the discovery of defects is less straightforward.

Research investigating the impacts of test-driven development to various such internal attributes has been carried out in the industry and academia alike. During the previous ten years or so, research has shown some of the attributes to be negatively or positively correlated with development done tests first but the results vary. In some cases, differences have been found between test-driven development practices and other methods of software development but the differences have not been large enough to be statistically significant as required by well-defined studies. Another issue to consider is that even if the differences are significant, are the measured absolute values above or below meaningful thresholds in practice? An analysis of the studies and the recorded results indicates the effects on particular internal attributes.

Defects

One purpose of software testing is to uncover hidden errors that have been able to somehow escape the watchful eye of the programmer or another individual that has been working on an associated product. With test-driven development, there should be a test for most of the program code but does this fact lead to a decreased amount of perceived defects? This topic has been covered in quite a few studies related to the development practice.

Early experiments from the industry hint that test-driven development could lead to reduction of defects. Maximilien and Williams report in 2003 of an experiment made at IBM where a distributed software development team consisting of somewhat junior developers utilized a test-driven approach in a project [MaW03]. The performance of the team was evaluated by comparing the defect rates from the project to

other similar projects within the company. On average, the defect density measured by the amount of defects and lines of code had been about eight defects per every thousand lines of code. Here, the team did better and was able to cut the defect density in half reaching a density of under four errors per thousand lines of code; in total the developed application had around 100 000 lines of source code including test code. Maximilien and Williams state that the improvement was likely due to test-driven design.

Williams continues her analysis of the IBM case with Maximilien and Vouk with more in-depth insights about the context of the industrial case study and acknowledges common threats to validity in case studies such as the difficulty of reproducing a project context that would be similar enough for comparison [WMV03]. The extended analysis puts more thought on defects and their origin. Interestingly enough, the breakdown of defects indicates that there is little change to the distribution of defects. Regardless of the development style, only a handful of defects were tagged critical and about a fourth of the defects were major; most defects were of medium or low severity.

IBM is not the only industry organization to report reduced defect levels with test-driven development. Nagappan et al. compare the results from the IBM study with similar case studies performed at Microsoft a few years later [NMB08]. Three experienced teams at Microsoft in different development divisions tried out writing tests first and their work was evaluated in relation to previous company projects. All of the projects lasted at least several months up to a year or so and were of moderate size with tens of thousands of source code lines although one of the projects was somewhat smaller.

Compared to the company baseline, the defect rates dropped: in the biggest project defect density decreased to a tenth. The results were similar in the other two as they showed a large reduction in defects and had less than a half the defect density of their baseline sister projects. Bhat and Nagappan describe that they tried to normalize the conditions of the case study by selecting the sister projects from the same manager but acknowledge that it might be difficult to derive decisive conclusions from such studies [BhN06]. Nevertheless, a tenfold quality improvement for the larger project can be considered significant and the three to four times better quality of the smaller projects at Microsoft is notable as well.

Industrial case studies give insight into the impact of particular development practices in a specific setting but there might not be much control over the variables

since the context changes from project to project. Controlled experiments on the other hand try to take the variables into account in the design of experiments. Defect rates have also been recorded in this kind of artificial settings and for instance Geras et al. write about their observations in 2004 [GSM04].

A handful of voluntary, yet experienced developers from the industry participated in an experiment that measured the effect of test-driven development. Geras et al. divided the developers to two groups which both developed a program in a traditional manner and then another one with tests first so that it was possible to validate whether the effects were due to the development practice. Besides developer tests, the experiment included acceptance tests that the developers were instructed to use in the development of both programs; it seems that the use of tests was encouraged in the test-last approach, too. The programming tasks were small and each task took only some hours to complete which is a short time compared to the industry case study projects that can go on for several months.

Defects in the experiment were counted in a unique fashion. Instead of looking up failure reports from a defect database, the experimenters had rigged JUnit so that the for each unit test the developers had to specify whether they *expect* the test to fail, i.e. were they working on the first stages of red-green-refactor and expecting a failure or running the tests at a later time when a failure shouldn't have happened. Thus, it was possible to record the number of unplanned failures.

There were only a few unplanned failures per each test case: the test-first approach was marginally better with most tests under two failures per test case and the test-last came in close with under three failures per test case. In practice, there wasn't much of a difference one way or the other and both development methods failed equally often with the higher level acceptance tests. It was noted that in general acceptance tests failed more often in an unexpected manner than unit tests which might result from the fact that developers were not accustomed to acceptance tests.

Aside from case studies in the industry, studies involving students have been made in academia that have contemplated defect rates and test-driven development. For instance, Vu et. al report in 2009 of a study where several developer teams took part in a student project which lasted for a year [VFS09]. During the project, the students developed software in three small groups that were instructed to either drive the development tests first or test after implementation. Not all the variables were controlled in the experiment, though: the teams developed software based on the same requirements but the programming languages and environments differed.

In the end, the team that was using test-driven design produced fewer defects than the team that was not writing tests first. Both project teams produced less than a hundred detected defects and the test-first team had an advantage of some twenty defects so the numbers are moderate overall. The other metrics from the study don't quite support this finding and the result remains inconclusive due to the different working habits of the teams. In fact, the team that was not assigned as a test-driven design team wrote almost ten times *more* test code than the team that was to concentrate on writing tests first; the intended team roles seemed to have inadvertently reversed in the process.

Students were also test subjects in another recent study that tried to find whether code inspections would help more in reducing defects than test-driven development [WNM11]. Wilkerson et al. conducted the experiment in a university with about 40 junior students who took part in a programming course. All students were briefly introduced to driving development with tests and the participants had the chance to practice writing tests in a short programming project prior to the start of the experiment.

Overall, it seems that the preparations of Wilkerson et al. were rigorous: students were randomized to different groups according to their scores in a programming test, code inspections were carried out according to a plan that considered biases of the order of inspections and various other validity threats were kept in mind when executing the study. For instance, it remains an open question how precisely the students in the test-driven group followed the doctrine of the researchers and actually wrote all unit tests before the implementation; most students replied to a survey that they were mostly able to do so and the high code coverage of the tests supports the claim of the students in this case.

During the experiment that lasted for a few weeks, the students were working as individual programmers and allocated to one of four different groups. The individuals in one group developed software so that their code was reviewed by a code inspection team and they were offered the chance to fix defects discovered by the inspectors. Members that belonged to another test subject group developed software against the same requirements but with test-driven development. A group that was using both code inspections and test-driven design was also formed to observe the combined effects of the two practices. Finally, a control group of students was used that neither utilized test-driven design nor code inspections in development. The hypothesis was that those using code inspections would be able to rid the software

of more defects than teams working with a test-driven design but the combination would be the most efficient defect reduction method.

The number of defects remaining in the final products was estimated by first running a set of automated unit tests that executed the code developed by each student. Test failures from these tests were added up by the amount of defects found by a separate inspection team that looked at the code of all participants.

Based on the raw results, the initial impression is that there are only marginal differences between the development methods with inspections leading the way and test-driven design left behind the other three. In the end product of around 500 to 600 lines of code, the results show an average of 12 defects remaining for the combined test-driven and inspection group and 15 defects for the test-driven group which is slightly worse than the control group.

However, a closer statistical analysis suggests that there is a more significant gap between the groups. The gaps widen in favor of code inspections due to the fact that the students were unable to fix all the defects found by the inspection team and this seemingly decreased the reported defect reduction power of inspections. Statistical adjustments relating to group sizes and covariates had an effect as well to the analysis of the results.

With the adjustments in place, inspections and test-driven design put together appear to be superior with only half the defects of the control group which is now last on the list. There is a minor difference between the separate code inspection and test-driven groups so that it can be said that inspections are more efficient in defect reduction than development done in the test-driven fashion. The results indicate that compared to the last ranked control group and bare programming practices, test-driven development has an edge on defect reduction but the advantage is minuscule at best.

Code Coverage

Test-driven development involves writing automated unit tests that execute portions of program code. The written tests can be said to *cover* the rest of the code up to a degree. Coverage can for instance mean either *statement coverage* which equals to the number of source code statements executed by the tests in relation to the total amount of statements or *branch coverage* which indicates how many of the possible conditional branches are visited by the tests as there can be code that omits

certain branches altogether [PeY08]. *Block coverage* is alike statement coverage but concerns evaluating basic code blocks instead of individual statements. The higher the coverage of tests, the more thoroughly is the program tested which in theory should reduce the likelihood of defects.

In addition to statement, branch and block coverage, some studies have recorded the *Mutation Score Indicator* (MSI) which is another metric that can help to determine whether the tests recognize transformations of the original program. Mutation testing is a fault based testing method where errors are deliberately inserted in the original program by modifying operands, expressions and statements [PeY08]. If a test picks up the transformation and the test fails, the particular mutant is said to *die* whereas an unnoticed mutant *lives*. The adequacy of a particular test suite can be evaluated by looking at the number of mutants that live in relation to the total amount of mutants.

Industry projects that have been using test-driven design have in some cases enjoyed a high coverage. Nagappan et al. report on the block coverage levels of the IBM and Microsoft projects detailed earlier [NMB08]. At IBM, the tests written by the team covered 95 percent of the code blocks while at Microsoft the smaller projects achieved a good coverage of around 80 to 90 percent. Only the largest of Microsoft's projects had a lower block coverage of 62 percent which means that a bit over a third of the code was not covered by tests.

Industry professionals have shown the capability to write test code that covers a large portion of the program in other cases as well. George and Williams formed programming pairs in three different companies who were given the task to program a small application in a day using pair programming and test-driven principles [GeW03]. Although short, the code that the expert programmers wrote covered a good 90 percent of statements and branches.

Code coverage of tests in industry projects has also been studied over longer periods of time. Janzen and Saiedian conducted a series of experiments in a longitudinal study where several groups of senior developers from the same company were tracked throughout multiple projects that used different development and implementation techniques [JaS08]. Albeit coverage was only one of the variables tracked, it's an interesting notion that the behavior of developers might be affected much by the type of work they've been doing recently.

In the set of experiments, teams were doing multi-month projects with one technique and then switching to another; it seems the teams carried their experiences with test-

driven design to subsequent projects. During the first stage, a team at the company started on a project working in the traditional test-last manner and finished off the project with test-driven design. The team created no automated tests in the beginning so the statement or line coverage was nil for the test-last section but picked up the practice quickly and ended up having almost a full code coverage with tests. Next, a team completed two *different* projects in the same order as previously and this time the coverage of the test-last project had increased to about 30 percent while the test-first project kept a substantially high over 95 percent line coverage. However, the maturation effect shows best in the last experiment where the team *first* worked on a project using test-driven design and *then* switched to a test-last approach: both projects had a high line coverage but this time the test-last project had a higher coverage of over 80 percent or so against the 70 percent of the test-first project.

Compared to the company baseline data collected from a wider array of projects, the results from the experiment vary. Over the time span of five years, projects using test-last development methods achieved a coverage rating of 40 percent on average and those using test-driven design were able to double the coverage. At least in this case it can be noted that test-driven design seems to increase the coverage of tests written.

Janzen and Saiedian focused their investigation on students, too. The first batch of students consisted of junior students who were briefly introduced to test-driven design and their performance and products were compared against those of other students in a summer project [JaS06]. For students with limited prior experience, the setting turned out to be difficult and the groups couldn't quite stick to their assigned development methods.

The line coverage of the test-driven group was down to 20 percent or so and the group considered to represent the non-incremental design had a better line coverage by 10 percent. The branch coverage of the test-driven group was slightly better at 40 percent, twice as much as the group which wrote the tests after implementation. In contrast to the almost full coverage of the test-driven industrial teams, the coverage can be considered relatively low. When the same student experiment was repeated with more senior graduate students, they fared much better in terms of coverage and closed in on the numbers of the industrial teams [JaS08].

The students' ability to adhere to test-driven practices leaves room for reflection. Müller and Höfer continue this line of thought in their study where they try to

analyze the differences students and industry experts have when they do test-driven development [MüH07]. Judging from the results of the study, students and experts might think in a different way when trying to apply a test-driven design pattern.

If the definition of the development practice is that no code whatsoever should be written without a failing test, even the experts are not perfect in this sense. When looking at all the code changes made, the experts wrote a bit over 80 percent of the code in such a manner that can be said to be test-driven design. But here is the difference: the students wrote less than 70 percent of the code following the principles and the variance between students was great; students struggle more with adherence. The skills of the experts also show in that they were also able to write the code using shorter development cycles and in some cases a bit faster. In this study, the code coverage levels were very high for both groups, though: statement and block coverage were over 90 percent but for block coverage there was much greater variance in the student group.

Students have also been involved in more controlled experiments that have studied the effects of test-driven design and the impact of the development method has not always been significant when looking at the results. Madeyski organized an experiment as a part of a university course where senior students were randomly assigned to test-first or test-last groups [Mad10]. Initially, it seemed that the test-driven group excelled and the branch coverage was slightly better than of the other group: means of both groups were in the range of 60 percent but the variance was smaller inside the group that wrote code tests first.

However, when looking closer at the results, the first impression turned out to be false in this case and the confidence didn't hold in the end. Before the experiment began, the participants had to answer a set of questions that tried to pin down their level of experience, for instance. An extensive statistical analysis of the experience covariates proved valuable; it was not likely that the difference between the groups was due to the development method but more likely due to the differences of individual experience in programming. Personal skill in developing enterprise applications and previous grades of the students simply seemed to matter more than the use of test-driven design.

Besides branch coverage the same applied to the mutation score indicator that was under scrutiny as well in the experiment. The indicator was considered in addition to branch coverage as it was seen to yield a better overview of the ability of the tests written. Even before the covariate analysis there was little difference between

the two development methods in the mutation score indicator. The tests written by both groups were on average able to handle a bit less than twenty percent of the total transformations generated by the use of mutation operations on the original program code.

Recently, there have been other works that have evaluated the effectiveness of test suites by branch coverage and mutation testing in an academic environment. The interest of Pančur and Ciglarič was to observe the impact of test-driven design to various quality factors in a university course where the senior students were first taught the process after which the actual experiment took place [PaC11].

Individual students were randomly assigned to groups of test-first or test-last developers but students in both groups also performed pair programming. The idea behind the experiment was not only to compare the two distinct development methods but also to see whether the supposed quality benefits of test-driven design are due to short development cycles which were used in both cases or actually due to the method of writing and thinking about the test first. The several month experiment with slightly different projects was carried out twice in the period of two academic years.

At first glance, the results on coverage seem to strengthen the view that it's possible to attain a higher level of branch coverage with a test-driven design pattern. But here too, a deeper analysis provides more insights to the actual effects. In the first set of projects, the branch coverage was actually worse when both individuals and pairs used a test-first approach but not by much. The longer pair programming exercise showed that pairs overall had a slight dip in branch coverage when compared to individual student developers but the coverage levels were good at around 90 percent or over. However, the student projects with the lowest branch coverage were omitted from the results as there was a certain threshold that was considered as an indicator whether the developers were able to follow the test-first principles. This resulted in uneven group sizes; several students chose to discontinue the experiment after a while which also contributed to the mortality rate.

The second iteration of the experiment the following year proved just the opposite and the tests covered more branches in the code when the developers were writing the tests first. While the around 80 percent branch coverage of the non-test-driven developers is not that bad, the 10 percent difference in favor of the test-first oriented developers can be considered meaningful to some extent even if the result is not statistically significant. But there's more to the analysis: the mutation test scores

turned out to be more controversial when analyzed together with branch coverage.

It seems that although developers in the first project were able to write tests that covered the application code equally well with both development methods, the quality of tests according to the mutation score indicator was not the same. There was a considerable difference in the scores so that the scores of the individuals that wrote tests before the implementation was much lower than the other developers that didn't do such incremental design. Somehow the test-first design and the high amount of tests didn't convert to the ability to handle adverse mutant transformations of the program code. Therefore the quality of tests written cannot be simply deduced from code coverage. A high coverage doesn't always lead to high quality tests that can detect atypical conditions in a program.

The role that test-driven development plays in relation to coverage and test quality remains somewhat unclear based on the results of the study of Pančur and Ciglarič. After all, the mutation score difference didn't exist in the second round of the experiment and the effects were inconclusive. A further statistical analysis revealed that although in some cases the numbers showed test-driven design to be effective, there was not enough statistical evidence to claim that the development method would be significantly better.

Complexity

To have a more elegant and simple program design is one of the objectives of test-driven development [Bec03]. The simplicity of program code can be evaluated by various complexity metrics such as McCabe's cyclomatic complexity or the Weighted Methods per Class of Chidamber and Kemerer. In light of these metrics, it is a fair question to pose whether simplicity is achieved through test-driven design.

The message from the industry is that in some cases the complexity might be somewhat lower when test-driven practices are applied. For example, Dogša and Batič were involved in case study where they looked into three different medium-sized projects from a telecommunications company [DoB11]. Two of the projects were test-last control projects and one was the project where the twelve team members were trained to use test-driven design in development.

All of the projects used a code framework as a basis for development and the teams developed the new code with unit tests on top of that. In total, the teams created over fifty thousand lines of new code per project in a development phase that lasted

more than a hundred working days for each project. Complexity was measured by the McCabe metric but in an unorthodox manner which ignored the relative complexity of components or individual methods. Instead, the complexities were summed up for all components in the code base that shows the overall aggregated complexity of the whole program. Comparison between projects of the same size can be meaningful with this kind of total complexity figure although complexity measures per method or class could have been more universal.

Nevertheless, the total complexity seems to vary according to the development method. All teams wrote a healthy amount of unit tests but the test-driven pilot group wrote even more test code. Whether it was due to the increased amount of tests or some other factor, the project that was making use of test-first principles had a lower total complexity than the control projects. Here, the difference was not quite double the complexity but not that far either; in the more modest case there were such structures in the code that caused twenty to thirty percent greater complexity.

The longitudinal multi-project industry study of Janzen and Saiedian recorded complexity metrics, too [JaS08]. In the first project, the code section that was written without test-driven design had twice more complex classes according to the Weighted Methods per Class metric ranging on average from 60 to 30 on the metric scale, respectively. The situation with individual methods was similar but the methods were not overly complex at all with an average cyclomatic complexity under four in case of the test-last section.

Perhaps the developers were more accustomed to writing simpler code and were able to draw on their experience but the following projects showed a gradual decrease in complexity when the teams were writing tests *after* the implementation. Classes no longer contained large chunks of methods and were less complex overall; in the last project the method complexities approached minimal sensible values. However, it needs to be noted that in the latter case the low complexities might have been caused by the hefty use of accessor methods that traditionally don't perform much computation and are thus simple by design.

Practically it seems that the complexities were pretty much on the same level if disregarding the results of the first project. Granted, the projects where test-driven development was applied occasionally had less complex code and even significantly so but the few cases show that it might as well be the other way around. Judging from the results of Janzen and Saiedian, test-driven driven design has the potential

to lead to cleaner code and on average the method of working appears to produce more consistent code complexity with fewer deviations from the standard but not always. The longer five year study of the various projects in the same company supports the idea that writing tests before implementation might have a slight edge on complexity: classes were thin and leaner albeit the methods only slightly simpler.

Students have on occasion followed the footsteps of industry experts when it comes to complexity but not consistently. Janzen and Saiedian's junior summer project students didn't quite follow test-driven design to the letter and so the recorded complexities are somewhat inconclusive [JaS06]. The student group that ended up writing tests only after the implementation of their program scored a bit better in terms of method complexity but the average complexities were overall low. A closer look at the program code of the test-first student group revealed interesting properties, though. The section of code that was not covered by any test turned out to be more complex so that untested classes had more complex methods and the classes were heavier as measured by the weighted methods metric.

Surprisingly, experienced students who took the same course didn't create lighter classes than the juniors [JaS08]. The classes programmed by the students in the test-driven group didn't have too many complex methods but on average the cyclomatic complexity was over five. Here, the test-last students did worse and the classes were big and bloated compared to the test-driven students. Albeit the students wrote only a handful of classes, the weighted method complexity was close to 160 (sic) and individual methods had complexity ratings over 6 which are by far the highest average values recorded by Janzen and Saiedian. This indicates that the test-driven students created a more sound design and the product of the test-last team was perhaps not too modular as there were many medium-sized methods in large classes. The weighted method complexity numbers do look devastating but in the end the capability to draw conclusions from this case is limited by the small number of code classes and students in the project.

In another context, undergraduate students that were to write tests before implementation showed ability to write simpler classes than a corresponding team of students assigned not to do test-driven development [VFS09]. Individual methods were more or less of the same complexity and on average the methods were just slightly more complex for the test-first team. But the test-driven design team created classes that had a simpler structure as there were not that many complex methods stacked in a *single* class. The latter result was of statistical significance although the numbers are

not that alarming in either case. There are some clouds over this result as well since according to the code coverage values, the process conformance of the test-driven student team seems low.

The more rigorous controlled student experiment of Pančur and Ciglarič offers additional viewpoints to the impact test-driven design might have on complexity [PaC11]. When individual student developers pushed the development effort ahead with tests, there was practically no difference whatsoever to the other developers who wrote their tests after implementation, even after statistical corrections. Only when students were paired up and used pair programming along with their assigned method of design and implementation, it was possible to make a distinction between the test-first and the test-last groups.

Pair programmers who used test-driven design did write code that was not as complex and their code had fewer outliers judged by McCabe's metric. Alas, in the second run of the experiment, test-last pair programmers had sections of code that reached complexity values of 138 which implies a rather complex structure with many branches somewhere in the code. The medians show that the two development methods were exactly not that far off and the conclusion is that although the complexity difference does exist, it is not statistically significant.

Thinking about complexity, it is good to bear in mind what McCabe originally noted about the structure of programs. He mentioned that well structured programs limit their complexity but those with cyclomatic complexity under ten are perfectly acceptable [McC76]. Times have changed since then but if this is considered as a sensible limit to complexity of individual methods then there are not that many cases which exceed the threshold. The highest reported *method* averages reported here with the students are close to six or seven on the complexity scale and only a handful go beyond ten. In the industrial setting with experts similar averages can at best be close to one. Hence, it would seem that methods written by software developers are mostly not all that complex. However, if code classes hold too many methods of some complexity, there might be a risk for the classes to become too heavy for comfort.

Coupling and Cohesion

As pointed out by some of the studies in complexity, in rare cases classes can have a few roles too many which can indicate a structural weakness. In particular, a class

can be considered incoherent if the methods do not operate on the same fields. The connections from the class to other classes can also indicate the presence of structural issues if the amount of relationships grows too large. It can be hypothesized that due to the incremental design strategy, classes written with the help of test-driven development could have fewer relationships with other classes and have a higher structural cohesiveness.

The relationships of classes can for instance be analyzed by the coupling metrics whereas the lack of cohesion metric can be used to study the integrity of a class. Janzen and Saiedian mention that it is not very easy to measure cohesion [JaS08] and certainly it would seem that cohesion metrics are included in rather few test-driven experiments. Likewise, there exactly isn't a plethora of studies where coupling would be a prominent metric.

Janzen and Saiedian did however include coupling and cohesion metrics in their experiments [JaS08]. The industry projects they observed generally had code and classes that were not too coupled. Classes written with a test-driven approach had between two to five connections to other classes but there were several cases where the coupling rose to over twenty which could be a hint of bad structure.

The projects that used a traditional test-last development method were not necessarily much worse than the test-first projects. In fact, the longitudinal industry study showed the test-driven code to have slightly *more* couplings than the code of the counterpart development method. Janzen and Saiedian report that the observed higher coupling might not be such a bad thing if it at the same time means that the classes properly use interfaces and other classes in their design.

The coupling values recorded in the student section of the experiment varied. Junior students who didn't follow test-driven design patterns so strictly wrote code that was three to four times more coupled than the code written in accordance to the test-last paradigm [JaS06]. The test-driven code that averaged over four and a half couplings per class also contained portions of highly coupled code that turned out to be associated with the graphical user interface of the program. When the code was further inspected, it seemed that classes which remained untested had the highest readings; code that was covered by tests and thus likely written with test-driven practices in mind was closer to the lower coupling level of the test-last code. The results from the later replicated study showed that another group of graduate students produced code that had a few couplings per class and that there was not much difference between the development methods [JaS08].

Normalized cohesion metrics from the studies indicated an unequal distribution of cohesiveness [JaS08]. The initial project which mixed the two development practices of writing no tests and writing tests before implementation demonstrated the no test code to be just a bit more tightly knit together. But the situation reversed in the following project where the test-first code was written so that the methods in classes were more often associated with each other through the fields but not much. In some projects, the gap between the cohesion metrics was quite big and in favor of test-driven design. In these especially apparent cases, the test-first code had twice the better cohesion than the test-last code. The result has some significance given the long five year period of the industrial study that was a part of the research.

The code of undergraduate students had a similar cohesion profile as the code in some of the industrial cases. So, the students wrote code that had actually less cohesion when they were driving the work with tests compared to the different orientation of not writing tests first. Again, not a big difference but notable. The more experienced students were able to create the cleanest code of all in terms of cohesion and most of the class methods accessed the same fields. A possible explanation for the good result is that the program only had a corresponding number of classes one can count with one hand. Since the results vary as much as they do, it cannot be stated without a doubt that test-driven design would always lead to classes which have a better cohesion and the same can be said for coupling.

Size

By definition, test-driven development requires practitioners to write tests. In an ideal situation, almost all statements in code are covered by some test in the test suite. This would imply that the codebase of the test-driven program would be larger than the equivalent of a program that uses only a few tests or no tests at all. Since measuring size is not tremendously difficile, there are a host of studies on test-driven design that have recorded the lines of code as part of their result set. Additionally, many of the studies make a distinction between the total amount of code lines and the amount of test code lines.

The theoretical maximum difference in size can be conceived by comparing the two extremes. Given that the two programs are functionally similar, obviously the program with the tests will be larger by the amount of test code lines. However, if the hypothesis is that the test-driven practice produces only the necessary functional code and no more, it can be considered that some sections of the code might actually

be smaller due to the mode of working.

Industrial case studies that mention code size are valuable as they offer a perspective on the actual work performed by industry experts. At the same time, absolute code size only characterizes the product and there is not much sense simply comparing the overall sizes as such. When available, it is better to compare the ratio of test code size to the non-test code size, i.e. to the code that actually defines the functional part of the program.

Maximilien and Williams report of the early efforts of the people at IBM who started using test-driven design as part of their development process [MaW03]. The initial report states that the distributed team developing the product wrote almost a line of test code for every two lines of functional source code. So in this case the ratio of functional source code to test code was 1:2 or more accurately 0.48. A later account of seemingly the same project points out that the ratio could even be higher [NMB08]; in the updated version, the size of the product is reported to be smaller containing some thirty thousand lines of test code and forty thousand lines of functional code with a ratio of 0.7. At any rate, the product is of moderate size and there is a good amount of test code involved which indicates that the team was able to mostly work according to the principles of test-driven development.

Microsoft is another large industrial company where case studies have been carried out. Bhat and Nagappan outline experiences from teams working with a relatively small Windows networking component and slightly larger communications product [BhN06]. In both of these projects, teams wrote a lot of tests: the test code to functional source code ratio was 0.66 for the networking project and 0.89 for the communications project. The latter means that the team wrote almost as many lines of test code as they wrote non-test code or in other words there was almost a line of test code for every line of normal code. At over twenty thousand lines, it begins to be a respectable amount of code.

The third Microsoft project reported by Nagappan et al. is considerably larger than the other two with a codebase size of over two hundred thousand lines of code [NMB08]. While the team had to write more code, they were able to maintain a decent level of test code. In the project, over sixty thousand lines of test code were written against the hundred and fifty thousand source code lines yielding the ratio of 0.39.

Dogša and Batič offer a more interesting angle from the industry [DoB11]. The setup is of higher interest as the case study has a setting where two projects were developed

with an iterative test-last practice and one project with a test-first practice. All of the projects contained code worth over fifty thousand lines of which the quantity of unit test code varied between thirteen to nineteen thousand lines. As might have been expected, the test-last projects were near the lower bound with ratios of 0.28 and 0.38 and the test-first project contained the most test code with the ratio of 0.49. The results are somewhat comparable as all three projects used the same code framework as a base and they were carried out with teams of the same size. Clearly, it seems that in an industrial environment with skilled experts, teams tend to write more tests when employing a test-driven approach to development.

Janzen and Saiedian go beyond the dichotomy of source code versus test code and analyze the size of individual classes and methods in their studies [JaS08]. In the first industry project which mixed test-driven design and development without tests, the classes written in the test-first stage were smaller by half and the methods were much shorter although the method count inside classes was more or less the same. The class size average close to eighty lines of code seems large compared to the tad over twenty lines of the test-first driven classes. Likewise, the difference of fifteen code lines for each method is notable as most methods in the project were very small. It is as if the methods are almost too small to contain a sensible amount of functional code since the five line code average of the test-driven methods doesn't leave too much room for anything but the bare minimum.

Consequent industrial projects observed by Janzen and Saiedian didn't quite show impact of the same magnitude. Classes were still somewhat smaller when teams were working with test-driven practices but the results in respect to size were not that far apart when the same teams switched to the test-last development mode. Although classes overall were smaller, single methods were even slightly bigger with test-driven design; mostly, methods in either case had a tiny size of five lines of code or less. Janzen and Saiedian report that considering method sizes, the supposed advantage of the test-last development practice was in fact likely due to the heavy use of accessor methods which are very simple by design and contain only a line of code or so. All in all, the projects were not large enough to make statistically significant observations but the longitudinal five year follow-up industrial study hints that classes and methods could be smaller when developers apply the test-driven principles when designing and implementing the code.

An overview of the size of code in student projects allows to evaluate whether experience or other factors like the academic environment could have a considerable

effect on the size of code elements. The undergraduate students that participated in the first iteration of the study of Janzen and Saiedian wrote code which had classes and methods of moderate size [JaS06]. The group of students who were trying to follow test-first practices wrote a bit bigger methods than the students who wrote the tests after the functional implementation and the largest methods were created by students who wrote no tests at all. However, the relatively low adherence to test-driven design is visible from the lines of code metrics. With around a thousand lines of source code, the test-driven students wrote less than two hundred lines of test code, yielding the test code to source code ratio of 0.16 which is around the same range achieved by the students who wrote tests in the end. Compared to the industry standard level, the size of test code seems small.

In the second iteration, students observed by Janzen and Saiedian were able to follow the test-driven process with more precision [JaS08]. Classes and methods written by the more experienced students were actually larger than those written by the junior students so at least experience is not the only factor that can influence code size. But there was quite a contrast between those graduate students that were taking advantage of test-driven design and those who didn't.

For the same requirements, the students tagged as test-last created far fewer classes which were over a double in size with 140 lines of code and above. The large classes also had methods which were of greater size and there were substantially more methods in the classes compared to the ones constructed by test-driven students. Intuitively, methods that have just over thirty lines of code don't seem excessively big but if the results are put side-to-side with the industry cases and the previous student experiment, the product created *without* test-driven practices by the graduate students seems to be on the heavy side. Janzen and Saiedian conclude that despite the relative differences, the numbers from student studies don't provide enough evidence to draw statistically significant conclusions.

The outcome of the study of Vu et al. [VFS09] bore slight resemblance to the student experiment of Janzen et al. [JaS06] which involved junior students. Alike, students who were instructed to use test-driven development wrote a limited amount of tests and were able to reach the source code to test code ratio of 0.17 that is considered not to be too high. The results were the same for both of the test-driven student teams. Surprisingly, in this case the students who were to write tests last fared better. Much better.

The roles in the student teams got reversed somewhere in the process of development.

Vu et al. write that as the project came to a close, the test-last student team was coding tests in a frenzy-like state to attain a better code coverage for their tests. This shows especially in the code size metrics: the test-last team wrote *more* test code than they did normal source code with the ratio of 1.2. The four thousand line test suite is of some size and the ratio exceeds even the industrial cases covered here. It seems naturally that the amount of test code created is correlated with the conformance to the test-driven process; test code will not get written if the role of tests is weak in the development process as acknowledged by Vu et al. in their report.

Madeyski and Szała conducted an academic experiment which allowed to assess the performance of an industry expert under somewhat controlled conditions [MaS07]. The behavior of the expert was monitored in a single project that was divided into three development stages: the first stage was completed in the manner that tests were written after the implementation, the second stage using test-driven design and finally the third stage was carried out in the same way as the first one. There might be a maturation effect as a single programmer was responsible for all the three stages and the sample size is small to reach statistically significant conclusions. Yet, there was some difference in size between the development stages.

User stories that were implemented in the middle of the project with a test-driven approach tended to have a smaller code footprint than the stories in the other two stages; at most, the average difference was around fifty lines of code per user story. According to Madeyski and Szała it could imply that there is a tendency for the code products to be smaller when developers are working in a test-driven fashion. But it is to be noted that there is quite a variation in the code size of implemented stories as there were fewer code lines per user story in the last stage than the first although the code was written using the same design and implementation method.

While performing another experiment, this time with students, Madeyski recorded the amount of test and production code written by the students who were divided into two development groups in which individuals either used test-driven design or practiced a test-last development process [Mad10]. The performance of the students was closely followed and their adherence to test-driven development was checked by inspecting the programming events gathered by an additional monitoring component. Those who didn't comply with the process to a sufficient extent were not included in the analysis.

Perhaps partly due to the emphasis put on conformance, the students wrote a good

amount of test code. It seems that the results do not distinguish between the two groups when it comes to code size but the average test code to production code ratio of 0.84 is quite high, especially if the figure also includes those students who were not using test-driven design. Each project consisted of a few thousand source code lines so even if slightly on the small side, the ratio is up to par of the industry and suggests that the amount of test code lines can approach the amount of other code written if the test-driven philosophy is internalized by the developers.

Müller and Höfer's study is of interest as it involves students *and* experts who were to develop a program that had the same requirements for both groups [MüH07]. Rather than comparing two distinct development methods, the study focused exclusively on test-driven development and on differences that might exist between people with varying levels of experience. One of the initial hypotheses was that professionals with greater skill could write more compact programs and test code than the students with less experience but this didn't entirely turn out to be true.

Experts were in fact capable of writing program code that had fewer statements. The test code, however, was of about the same size; experts were just as verbose as students when writing tests. Even if some parts of the program code were shorter, the amount of lines changed throughout the development process was equal. This means that experts and students did around the same number of changes to both the program source code and the test code. So theoretically there might be the possibility that experience leads to better understanding of the programming activity and shorter code on some level but the evidence is not statistically significant. Perhaps it could be hypothesized on these grounds that test code would be less demanding to write than other source code if the presumed ability of the experts is shown on one side but not the other.

Effort

Test-driven development provides a frame for the development activities of a program and as a result programmers seem to put a lot of effort in writing program code for the automated tests in addition to the standard source code that makes up the functionality of the program. Considering the test code to source code ratios of 1:2 or higher, this could mean that the end product with the tests is one and a half times larger than the one developed without any tests. Writing the extra code lines takes time so the hypothesis is that it would take more effort to write a program with test-driven design than without.

The comparison is by no means straightforward. Although tests have a central role, they are just means to an end as Beck writes in his book [Bec03]; the tests drive the design and the way developers work. Reflecting on this, it might not be such bad thing if developing takes more time with test-driven development if at the same time the methodology helps to organize the related activities more sensibly and structure thought of the developers. Furthermore, given a set of tests that cover much of the source code, the effort spent on testing might be reduced.

This section concentrates on the results of industrial and academic studies that have placed the effort spent on development under the loop. The studies help to give insight into the question whether test-driven design consumes more time than other development methods. Measurement of development effort itself is direct and the activities can be clocked by any chronograph from which time can be observed: the unit can be anything ranging from hours to person months depending on the type and length of the study.

Introducing test-driven development into industry organizations and software development teams is a process that can take a long time. Crispin writes from her experience that it can take professional developers years to adapt to the new way of thinking about tests first [Cri06]. If the adaption is indeed as challenging as described, measuring effort in short pilot projects might be especially disadvantageous to the process of test-driven design. Furthermore, if experienced industry professionals are having a hard time in getting to know the art of writing good tests, students facing a similar scenario in the academic experiments might not be less perplexed about the situation.

George and Williams arranged an early experiment in 2003 with experts from three different companies [GeW03]. The enlisted professionals were not completely accustomed to test-driven design but some of them had previous experience on the development method. Development in the experiment was carried out in pairs with the objective of creating a game either with test-driven development or by conventional means without many tests. The other half that was driving their development with tests turned out to be slower by about sixteen percent in finishing their projects so the development without tests was more rapid. The extra development effort is on the same level as reported in some of the later studies in industry organizations [NMB08].

Nagappan et al. summarize experiences from industry projects at IBM and Microsoft where test-driven development had been taken into use by a number of teams

[NMB08]. The various size projects were compared to other projects completed earlier inside the same organizations. Development effort itself was not actually directly measured but managers estimated that at IBM the project took over fifteen percent longer due to the test-oriented method of working. Likewise, managers at Microsoft estimated that the development time of their projects increased by at least the same factor and the estimates for some of the more time consuming projects stated effort increases from twenty to thirty percent. Nagappan et al. emphasize that the estimated increase in development effort is not seen as negative since in these cases test-driven design lead to products that had fewer defects relative to the standard of the respective organizations.

In addition to some of the longer projects, experts from the industry have also participated in short programming assignments—lasting just for a few hours—where the effect of writing tests have been evaluated. Canfora et al. observed of 28 programming professionals in this kind of a factorial experiment where an application was made by the same people with two different implementation techniques of test-driven design and testing after coding [CCA06]. The interaction effect of the setting to the total effort seemed to be quite clear; when programmers were using test-driven design, it took them about one and a half times longer to finish their assignments compared to the setting where no tests were written. There was enough data to claim statistical significance of the results which, for instance, support the findings of Microsoft and IBM managers who based their opinion about effort on estimates.

However, there are experiments with professionals which haven't shown such drastic differences in effort. Considering the setup of the experiment, the setting in the study of Geras et al. [GSM04] resembled the one of Canfora et al. as both studies had two alike treatments and the focus was on professionals whose work was assessed. The time spent on programming was compared to a pre-experiment effort estimate given by a number of other experts; the difference between the actual and the estimated effort was discussed in the study. In both treatments, with and without a test-driven process, the programmers exceeded the estimates but the effort medians between the groups were close. At least test-first programming wasn't found to be slower; on the contrary, the results showed greater predictability for test-driven design in the form of less variance in effort.

One of the interesting questions related to effort and test-driven development is the relationship between the time it takes to develop a product and the time it takes to maintain it. Fortunately, this is precisely what Dogša and Batič examined among

other attributes: the industry projects had a span of several hundred development days and the subsequent maintenance period of about equal length was followed by the authors [DoB11]. Two out of the three similar projects worked in such fashion that the programmers wrote tests after the implementation and in one of the projects programmers utilized test-driven design. Indeed, there was a noticeable difference between the two methods.

Development did take longer when the programmers applied test-driven principles in their work but there is more to the results than meets the eye. When the products were finally released, the test-driven project had taken considerably longer to implement: it was completed thirty to forty calendar days after the two others with three to four thousand *more* working hours spent compared to the other two projects.

However, in the maintenance phase the scale tipped. Those maintaining the test-driven project were able to service maintenance requests faster on average—by thirty percent or so—than the people from the non-test-driven projects. This is to say that the average maintenance effort, an indirect metric, calculated from the time spent on maintenance and the number of maintenance requests over a 260 day period was considerably smaller.

When summing up the effort from the development and maintenance stages it was seen that test-driven development was still a few thousand person hours behind at the time when the observation ended but the gap narrowed with every maintenance request. Hence, by the results reported by Dogša and Batič it looks that projects can be more maintainable when test-driven development is used but it takes time to realize the accompanied benefits.

The effort profiles yielded from experiments in classroom conditions have not always pointed to the same direction as the industrial case studies; while most industry studies report of increased effort, some experiments on students find similar effort levels despite of the development method or even proclaim test-driven design to be the faster implementation method. For instance, two teams of students in the experiment of Janzen and Saiedian spent a roughly equal time in developing the application but the test-driven team completed more features and in the end used up more time in the course of the project [JaS06]. The third team in the experiment that neither wrote tests before nor after implementation reported of effort equal to the two other teams combined; a result which might not fully capture the actual differences between the different implementation methods.

Students in the study of Gupta and Jalote showed that concerning effort, test-

driven development can outperform traditional ways of development where testing is mostly manual and up-front design is more extensive [GuJ07]. The arrangements of the study were such that during three weeks, the students completed two different applications. First, the students followed either the test-driven or the traditional process as assigned and upon completing the first stage switched to the other process flow for the second part of the experiment.

Students whose work was driven by tests were able to kick-off the implementation phase sooner than the students who had to plan their program more thoroughly in the beginning. This property seemed to be true in both projects and the design-oriented students spent three times longer drafting their plans. For the overall effort, the differences were less considerable but still of significance in the other project.

Consequently, test-driven student developers didn't spend as much time with their project as did those students developers that planned their activities with more care first and then implemented their design. Again, this rather surprising trait was observable from the two projects in the experiment and in one of the projects the completion times were far enough apart to claim statistical significance. In the project that showed statistical significance, it took longer for the test-driven students to test their product during the development phase but they spent less time fixing defects later; it can be concluded that the effort difference in the experiment stems from the speedier project ramp up time and the reduced time it took to fix defects. However, this result is not entirely universal as in the other project the people who didn't write tests used more time on testing in the development phase. Still, given the right circumstances, it's possible that test-driven development can be just as rapid as traditional software development if not a bit faster.

Huang and Holcombe hold the same opinion that students tend to use more effort on testing when a test-first development process is in place [HuH09]. The experiment of Huang and Holcombe—which the result is based on—took place in a software laboratory where students were observed for twelve weeks in various projects. Half of the students who were involved in test-driven teams spent between five and ten percent of the whole project time on testing activities whereas those students who were developing in the traditional manner where tests come last spent less than five percent of their time on testing. It is noted that some of the increased effort might be due to the fact that the test-driven team experienced the impact of *rework* i.e. the tests that had already been written had to be re-engineered when the requirements changed. This could be a sign that in environments where the requirements are in

a state of flux, test-driven teams might need to throw away more code than teams that don't write so many tests.

In addition, Huang and Holcombe report that there exists an association between the testing and programming efforts when looking at test-driven student teams. The hypothesis was verified by analyzing the data of previous student projects as well since there wasn't enough data in the original projects to make concrete statements about the property. According to this property, whenever much effort is put on testing, less time tends to be allocated to programming activities. The result seems rather natural as distribution of finite time regulates the possibilities of allocating time so that if effort increases in one activity it must decrease in another. However, the correlation is more complex: the hypothesis didn't hold for students who were *not* following a test-first strategy. So if the test-last students expended time on testing, their programming workload didn't reduce like it did with the test-driven teams. Huang and Holcombe state that the reason behind this fact might be the way the work of test-last teams is organized; testing doesn't directly control the programming effort if tests are written in a late stage of the project or not at all.

Finally, it is appropriate to consider the experience of subjects when assessing the effective impact test-driven development has to the duration of the project and the effort required by the subjects to complete the project. In particular, Müller and Höfer noticed that students who were unaccustomed to test-driven design and had little industry experience couldn't quite match the pace of the industry experts who were more mature as programmers [MüH07].

In this short experiment where individuals from both groups used test-driven design, the experts were significantly faster in completing a specific experimental phase; the median of the experts was around 160 minutes whereas half of students spent more than 250 minutes working with the task. Reflecting on this, the results from industry studies might not be easily comparable to the results of academic experiments. It is likely, though, that as the young programmers gain knowledge in software development, they'll be able to apply the principles of test-driven design more rapidly and thus finish projects faster. However, as shown by the industry cases, test-driven development might not reduce the overall project effort in the short term.

5.3 On the Effects on External Attributes

External attributes that are associated with the environment more than internal attributes are difficult to measure and there can even be discord concerning the definition of external attributes such as external quality [FeP97]. Perhaps this is one of the reasons why studies that focus on test-driven development seldom concentrate on external attributes. But few do so and besides general external quality discuss attributes of productivity, cost and maintainability. This section describes the results of some of the studies that cover external attributes.

When measurement becomes too ambiguous or is otherwise not feasible for a particular dimension of quality, one option to acquire information is to ask around and perform various surveys. Dogša and Batič did just that and interviewed project participants in their industry study and asked how they felt about test-driven development as opposed to traditional development where tests are written last [DoB11]. Many developers felt that test-driven design took more time from them but it was seen as a pathway to better quality and more sound code. As for maintainability, every developer who answered the particular question saw test-driven design to increase maintainability; although, the question was closed—the only possible choices were *yes* or *no* for the question.

In another survey that was part of the study by Gupta and Jalote, student developers were more insecure of test-driven practices [GuJ07]. According to the survey, students were less comfortable with the product they developed through incremental test-driven design and subsequently had more faith in the design of the product managed through the traditional development process. Still, even if uncertain of the overall design, the students did feel that they were able to achieve a higher standard of testing by applying test-driven principles. The result of the questionnaire would imply that in the minds of the students, test-driven development leads to higher testability of the product.

Questions about the perceived quality of a project and its associated products can be posed to the people for whom the software is being developed, too. Instead of asking the opinion of the developers in a student experiment with multiple projects, Huang and Holcombe went directly to the project customers and asked what did they think of the quality of the work produced by the students [HuH09]. In the survey, quality was broken down into ten different dimensions ranging from documentation to the reliability of the system in use without forgetting aspects of functionality and usability.

The answers were collected after the customers had been able to use their student-implemented systems for a month and the scores from each question were summed up for analysis. Roughly, the customers didn't see major differences in quality between those projects that were implemented by students who were using test-driven processes and those who were not: the scores were equal. In fact, some project customers felt that the systems developed in the old fashioned way—with tests written late in development—were of higher quality. The result was against the initial hypothesis of Huang and Holcombe; they expected software produced with test-driven development to have better quality. The relative inexperience of the students and the focus of testing to too small units were seen as potential reasons why the test-driven teams failed to deliver superior quality. To conclude: test-driven design might have a positive impact on some attributes as seen by the developers in the other studies but from the viewpoint of the customer it might be harder to see all the benefits.

Productivity

Productivity is an external resource attribute which means that the output level of teams and individuals can be observed to some extent in a given environment [FeP97]. It is an indirect attribute where typically the size of the product is divided by the time used to make the product: often lines of code per hour or features per hour. Although there might not be much sense in drawing concrete conclusions from productivity figures [Gla03], productivity seems to be quite well covered in rather many studies on test-driven development. If the message is—especially from the industry—that the products are bigger as more source code lines are being written in the form of tests *and* the development takes a longer time, then this could suggest that productivity of teams and individuals is lower if they're using a test-driven development process.

Dogša and Batič had research questions about productivity in mind when they were conducting their industrial case study [DoB11]. Finishing three different projects took around twenty thousand person hours each and a considerable amount of code was produced during this time. There was a noticeable difference in productivity between the projects. In the two projects where developers utilized the familiar process of writing tests after the implementation, productivity was reported to be over two lines of code per person hour. Productivity was not as great for the developers in the project where developers used test-driven design; on average it was

less than two lines of code per person hour. Lower productivity was anticipated by the researchers and the hypothesis realized in the study. When asked, developers also thought that the cause for the lower productivity was specifically the laborious test-driven development process.

There are some cases where professionals have been shown to work faster when they have applied test-driven principles. Madeyski and Szała focused on productivity in an experiment where a professional was recruited to develop a project in several stages [MaS07]. Besides using lines of code for measuring productivity, the experiment recorded productivity ratings in terms of implemented user stories and passed acceptance tests per person hour.

In the first of the three stages, the productivity of the developer was a moderate 25 lines of code per hour when the process was fixed to a test-last approach. However, in the next stage, the pace of the developer nearly doubled when using test-driven development and this showed also in the number of passed acceptance tests. Interestingly enough, when the developer switched back to the test-last method in the final stage, the productivity figures stayed on the same higher level as in the intermediate stage. Thus, although there is some claim to increased productivity, it remains somewhat unclear whether the higher productivity could be attributed to the process in use. Madeyski and Szała also agree that the small sample size might not give enough statistical power to the result.

Students have also shown higher productivity in certain studies when they have been using test-driven development. For instance, the test-driven student subjects in the study of Janzen and Saiedian had quadruple productivity of around 28 lines of code per hour compared to the teams which followed other process models; a fact slightly shadowed by the low process conformance of the test-driven teams. Likewise, Gupta and Jalote found those students that developed their program the test-driven way to be a bit more productive than the conventional test-last students [GuJ07]. The finding is not statistically significant but in one of the projects half of the test-driven students had a productivity of around 60 lines of code per hour or over while at the same time half of the conventional process students scored under 50 lines of code per hour. The productivity ratings were closer to each other in the other project that was part of the study of Gupta and Jalote but at least there might be some conditions under which test-driven development is not significantly slower.

In a number of other studies, students have *au contraire* been slower to develop their programs when they have been subjected to the practices of test-driven development.

Desai et al. noticed that fresh students who were taking early programming courses spent more time with their projects when they were encouraged by grade incentives to write tests first [DJC09]. Thus these students were less productive than students who also wrote tests but didn't have a similar external reward incentive offered to them.

Along the same lines: in the study of Vu et al., the student team not powered by tests was twice as productive as the two other teams instructed to use the test-first approach [VFS09]. Productivity in this case meant that the test-last team finished two times as many features and also the team's lines of code per person hour was better by the same factor. It must be noted that that the team roles didn't quite hold and test code didn't always precede functional code for the test-first teams.

The results on productivity can be considered to conflict even in similar environments where people have a comparable amount of experience; several high-rigor student experiments have shown that test-driven development might not have an effect on productivity at all. Among other attributes, Pančur and Ciglarič studied student productivity with different development processes over the course of a few years with a focus in individual and pair programming [PaC11]. Regardless of the development process, the amount of implemented user stories per person hour—used as a measure of productivity—didn't seem to differ that much. Productivity was practically identical especially in the more time consuming pair programming section of the experiment and the students implemented as many user stories with an iterative test-last process as with a test-driven process. Student programmers were actually slightly more effective as individuals when they were working according to the test-driven process in the shorter section of the experiment but the figures were still close to each other and the results were found to be far from statistical significance.

Huang and Holcombe eventually came to the same conclusion that evidence doesn't entirely support the existence of differences in student productivity that could be linked to the method of development [HuH09]. The initial hypothesis in the student experiment of Huang and Holcombe was that test-driven teams would have a higher code output rate and that productivity would increase when more time would be allocated to writing tests. The first hypothesis didn't turn out to be completely without merit since on average the productivity of the test-driven teams was a bit over twelve lines of code per person hour opposed to the seven lines of code per person hour of the test-last teams. However, the ten student team sample size

didn't give the experiment enough statistical power to make the claim and the teams that were writing tests first had a higher variance in productivity. The variance was reported possibly to originate from the short time the students had been acting as programmers altogether and the limited experience the students had with test-driven development. As for the second hypothesis, data from the experiment indicated that productivity didn't change with test effort so the extra time spent on writing tests didn't reduce productivity. This makes sense since unit tests also contribute to the size of the product if measured in lines of code and if the test code is included in the evaluation of productivity. It is uncertain, though, whether the productivity figures from this experiment included test code in the calculation of the productivity metric.

Concerning productivity, the background of the developer seems to matter quite a lot and thus it might not be fair to compare the productivity of students head-to-head with the productivity of experts. According to the study of Müller and Höfer, there is considerable distance between the output rates of the two groups [MüH07]. For production code, the experts were able to produce over 50 lines of code per hour in the experiment whereas most students were able to reach a rate half of that; both experts and students were instructed to use a test-driven process.

An interesting fact was that developers *didn't* write functional code with the same speed as they wrote test code. Both groups picked up a tremendous amount of speed when they were writing the code for the unit tests. Experts were still faster but they were thrice as productive with most reaching a rate of 150 lines of code per person hour and likewise the students were able to write more than 100 lines of test code per hour. It appears that people in general are quicker to write unit test code and thus it could be argued that test code is simpler to write than the functional code that implements the requirements of the test. This result could have some implications to the way productivity is understood in projects that use test-driven development; producing a line of test code takes a different amount of time than producing a line of functional code and this is something to consider.

Cost

Cost is a non-functional requirement and a development constraint in software engineering [Lam09] so economical incentives can be argued to guide decision making and direct action. Therefore it is of some interest whether the application of test-driven development leads to a *reduction* or *increase* in cost. The increased size of the products coupled with the extra effort required to write tests would suggest the

initial hypothesis to be that test-driven development would also be more costly.

Dogša and Batič didn't study cost as such but the reported time to market figures of the industrial projects in the study puts weight to the matter [DoB11]. It can be considered consequential that the test-driven project delivered over thirty calendar days later than the other two similar projects which were completed in a bit under two hundred calendar days; the development costs must have been higher for the prolonged project. Then again maintenance of the test-driven code was found to be easier and less time consuming which would theoretically also lead to lower maintenance costs. Hence, the lifespan of the product in question should be considered when evaluating the costs involved.

Certainly there are simpler tasks than to infer the sources of cost in a software development project as cost is an indirect external attribute that can be considered specific to the environment. Still, Wilkerson et al. made an attempt to study whether test-driven development would carry a different cost than other development methods with a particular focus on code inspections [WNM11]. The metric used in measurement for cost was person hours.

On average, the students who were working according to test-driven principles were the most cost effective and used the least time implementing the experiment task. Then again, those students who were assigned to a control group were nearly as fast in finishing their task; people in the control group didn't use test-driven development or other specific development methods. The cost and effort went up when code inspections were added to the process and the combination of test-driven practices and code inspections carried the highest cost—three times higher than the test-driven process alone. Since scheduled code inspections require several experts to sit down and review the code, it would seem logical that processes including code inspections are more expensive than those processes where inspections are not used but the potential benefits of finding more defects with inspections should be weighed against the added costs. As Wilkerson et al. conclude: partly due to unknown factors development processes can vary in cost, at least when it comes to test-driven development and code inspections.

5.4 Summary

Empirical evidence about test-driven development flow from many sources and this literature survey has gathered information from a number of studies and experi-

ments. These studies have covered some aspects of software engineering related to the use of test-driven design principles in the development process. This section gives a brief overview of the publications that have been included in the survey.

Runeson and Höst name four research methodologies that are well suited to empirical research [RuH09]. In a *case study* the object of study is observed in its natural surroundings—the context—without much interference and thus the approach is good for exploratory purposes. A *survey* is a methodology where information is gathered from individuals or groups by posing a series of questions through interviews and alike. *Experiments* allow more control over the conditions of research as it is possible to manage the population of research and modify the treatment conditions as needed.

Because of the greater control involved, experiments can at best be classified as explanatory which means that relationships between the treatment conditions and the outcome can be suggested. In properly controlled experiments, randomization of test subjects is used to guarantee a distribution based on chance. When control of the experiment is not so strict and randomization is not used, the experiment is called a quasi-experiment.

Further beyond explanatory purposes, it is possible to use the *action research* methodology. Runeson and Höst mention that it is a dynamic form of research where changes to the research elements are common in the course of the research in order to improve some attributes that are being studied.

Origins of Research

The information about the effects of test-driven development for this qualitative literature survey originates from around twenty publications. To be precise, there are 19 publications that were selected for the review. The primary research methods which appear most often in the reviewed publications are controlled or quasi-controlled experiments but there are also reports of industrial case studies. Some of the reviewed studies use mixed research methods: for instance, a number of controlled experiments were augmented by surveys and interviews.

Most of the research has been carried out in academic environments either so that the subjects have been students who have participated in a course or then industry experts have somehow been studied in experiments conducted at academic research facilities. However, a handful of studies have been performed in real industry or-

ganizations. Table 5.4.1 lists the publications sorted by the context of the research and the name of the author.

Half of the publications are conference proceedings and half are journal articles. The majority of the articles and proceedings are affiliated with the Institute of Electrical and Electronics Engineers (IEEE) but many conference proceedings have also been published by the Association of Computer Machinery (ACM). In addition, a number of journal articles covered in this review come from the science publishers Springer and Elsevier.

Digital library services and search engines of the aforementioned respective parties were employed in the discovery of the conference proceedings and journal articles. The search strings entered to the search engines were such as *test driven development*, *test driven* or *test first* which initially returned a massive amount of publications; for instance, just the keyword *test-driven development* returned over forty thousand entries from the ACM digital library at the time of the writing.

Titles and abstracts of the several hundred highest rated publications of each search engine used were screened manually in an unsystematic fashion to find studies and experiments which would include information about the quality factors of test-driven development. Further pruning of publications was made based on the content and several publications were omitted due to restricted sample sizes or other experimental factors—the qualitative survey had to be limited to a reasonable amount of representative studies from the industry and academia.

Research Facets

There are many aspects of software quality and test-driven development that the research reports included in the literature survey tell about. Typically, the studies focus on not just one quality factor but on several factors at the same time; still more on the internal quality attributes rather than external. The emphasis on attributes varies, too: industry case studies might briefly mention an attribute such as size whereas a rigorous controlled experiment might analyze the same attribute more thoroughly. An overview of which publications cover which attributes can be seen from Table 5.4.1. An x is marked in the column of the attribute if the publication on the row has *some* data about the attribute.

While research has been conducted to understand the impact of test-driven development in the software development process and a multitude of attributes have been

Author Name and Year	Context	Defects	Coverage	Complexity	Coupling	Cohesion	Size	Effort	External Quality	Productivity	Maintainability	Cost
Bhat and Nagappan 2006 [BhN06]	Industry	x	x				x	x				
Canfora et al. 2006 [CCA06]	Industry							x_-		x_-		
Dogša and Batič 2011 [DoB11]	Industry	x^+		x			x	x_-	x^+	x_-	x^+	
George and Williams 2003 [GeW03]	Industry		x					x_-	x^+	x_-		
Geras et al. 2004 [GSM04]	Industry	x	x					x		x		
Maximilien and Williams 2003 [MaW03]	Industry	x^+					x					
Nagappan et al. 2008 [NMB08]	Industry	x^+	x				x	x_-				
Williams et al. 2003 [WMV03]	Industry	x^+					x					
Janzen and Saiedian 2008 [JaS08]	Industry/Academia		x	x^+	x	x	x^+					
Madeyski and Szała 2010 [MaS07]	Industry/Academia							x		x		
Müller and Höfer 2007 [MüH07]	Industry/Academia		x				x	x	x	x		
Desai et al. 2009 [DJC09]	Academia		x					x	x	x		
Gupta and Jalote 2007 [GuJ07]	Academia							x^+	x	x		
Huang and Holcombe 2009 [HuH09]	Academia							x_-	x	x		
Janzen and Saiedian 2006 [JaS06]	Academia		x	x	x		x	x	x	x		
Madeyski 2010 [Mad10]	Academia		x				x					
Pančur and Ciglarič 2011 [PaC11]	Academia		x	x					x	x		
Vu et al. 2009 [VFS09]	Academia	x	x	x^+			x	x	x	x		
Wilkerson et al. 2011 [WNM11]	Academia	x_-					x					x^+

Table 5.4.1: Research included in the literature survey cover a multitude of quality aspects regarding test-driven development in industrial and academic environments but not many claim statistical significance.

under observation, relatively few studies and experiments claim their findings to be statistically significant. Those publications where such a claim is made have been marked in Table 5.4.1 with either a superscript plus symbol x^+ for a positive effect or a subscript minus symbol x_- for a negative effect. A statistically significant positive effect means that test-driven development was seen to be better compared to some other development method used in the particular research; a development method without writing tests first was a popular object of comparison but not the only one used. A statistically significant negative effect means that subjects were worse off using test-driven development.

Regarding qualitative studies, for example industrial case studies, the significance indicator was also used even if no statistical analysis was made given that the research had qualitatively speaking enough confidence in the implication and that the result presented seemed indeed to be considerably better or worse. Furthermore, an

effect was considered statistically significant if the relevant null hypothesis regarding the attribute was rejected and the significance was suggested; positive or negative effects that were below the significance criteria set by the respective authors were not tagged as significant.

Taking a closer look at the attributes, effort seems to be the most popular and information about the attribute appears in 13 out of the 19 reviewed research reports. Here, the data is quite consistent and five studies and experiments find test-driven development to have a noticeable negative effect on effort: it takes more time to develop with a test-driven process and especially industry organizations seem to find such development time consuming. Against the five negatively affected attributes there is one study in which students spent significantly less time. Nevertheless, the negative effect of effort is also reflected in the results for productivity. Three out of twelve studies take note that test-driven development negatively affects productivity.

The impact on defects has been mostly studied in industry organizations but some academic experiments have recorded defects or failed amount of higher level tests, too. Although some of the industrial case studies in this survey partly report of the same cases, the trend seems to be that test-driven design has a positive effect on defects and thus the number of defects would reduce when the process is applied in an industry context. According to the study that reported a negative effect on defects, test-driven development is decent as a defect reduction method but no match for code inspections.

Size is an attribute that is featured in a good number of studies and experiments. There are a limited amount of studies that make concrete hypotheses about size itself. Rather, size is stated as a matter of fact to characterize the products in question and it has utility value in the sense that it can be used as a component for indirect attributes such as productivity and defect density. A study that did highlight size, reported of a smaller functional code footprint when a test-driven process was applied and the difference was considered significant. Several others didn't consider size to differ significantly. Still, industry case studies report of high test code to source code ratios which would imply greater size of the resulting products when test code is included in the evaluation of size; a concern which is overlooked in the statistical analysis.

Coverage appears as often as size in the research reports: 11 out of 19 publications contain information about the attribute. Good coverage of tests is seen as one of the indicators of proper test-driven process adherence, yet none of the eleven find

the effect on coverage to be statistically significant. Perhaps the increased coverage accompanied with writing more tests is seen somewhat self-evident and there is no need to hypothesize about coverage. It has to be noted that the correlation between the development method and coverage is not as clear as it seems. Experiments where coverage has been one of the key attributes analyzed have considered the effect on coverage minor—unit tests which cover sections of code are also used in other iterative development processes besides test-driven development and this narrows the gap between the development methods for this attribute.

External quality is more like a group of attributes rather than a single quantifiable attribute. The industrial case study and experiment which conclude external quality to be considerably better in test-driven processes than in alternative forms of development use the amount of passed acceptance tests as the metric for external quality. Whereas the same metric is used elsewhere for external quality, surveys to customers and other parties have been utilized as well. The common element is the external, environment specific, viewpoint which can yield information not easily attainable through internal product and process metrics.

Pure static code analysis is less frequent or at least in this literature survey the popularity of code structure attributes is relatively low. For complexity, in two out of five cases it is argued that test-driven development significantly helps to create less complex programs by reducing the complexity of code components. At least the other three don't consider complexity to increase so the trend seems to be positive. Attributes of coupling and cohesion are covered in one or two studies without notions of conclusive empirical evidence.

The rarest of the attributes are the external attributes of maintainability and cost which both appear once in the included research reports. Maintenance duties were reported to be notably less time consuming in an industry study when the project team had used test-driven development in the development phase. Likewise, a controlled experiment suggests that there could be a cost benefit involved when test-driven principles are followed.

Overall, most of the research results can be considered neutral or statistically inconclusive. The benefits and disadvantages of test-driven development are balanced: there are 12 occurrences of significant positive effects and 9 occurrences of significant negative effects. All the rest are neutral or inconclusive. The results indicate that test-driven development can have an impact on some aspects of software development.

6 Conclusions

Test-driven development interfaces with the core areas of software development and so has a chance to affect the outcome of the development process. Programmers that take advantage of test-driven design patterns alter the way the software is being pushed ahead with the incremental test-based design and implementation choices. The tests created in the process keep the piece of software in check and help programmers to embrace change by having a battery of rapidly-executed tests which can verify the correct operation of the system at any time. When there is need to change and evolve, the design of the program and the tests written can aid developers in adding new layers of functionality on top of the old ones or fix the existing layers with more ease. Only the areas of specification and validation are somewhat outside the influence of test-driven development and variants derived from test-driven development address these other concerns as well. The question remains: after reviewing the results, is the impact to these areas of software development worthy of note as judged by the empirical evidence and experience that has been gained from research surrounding test-driven development?

The qualitative literature survey that was performed as a part of this work reveals some of the possible effects that can be observed in industrial and academic environments when subjects are exposed to the practices of test-driven development. In the survey, data from a total of eleven different product, process and resource measures were collected and analyzed.

Concerning defects, the message from the industry studies is clear. A noticeable reduction of defects associated with test-driven development was reported in the majority of industrial studies. Research-wise, the industrial case studies provide challenging grounds; in the murky waters of the industrial environments, the causalities are not so clearly visible. Still, it is hard to put aside evidence which describe reductions of the magnitude discussed in the studies. Studies in the academia don't entirely reflect the same phenomena but effects might change with the conditions. The finding that test-driven development might lead to reduced defects is in accordance with the analysis of Shull et al. [SMT10].

The impact of test-driven development on the internal quality of program code was one of the aspects in the survey and it seems that the results are partly indifferent. If the objective of test-driven design is to have code that is simpler and less complex, then the objective is achieved to some extent. There are studies which

support the idea of having methods with fewer branches and subsequently classes with fewer complex methods. It is unclear, however, how complex the code written by skilled programmers really is and do the relative complexity differences between development methods affect cognitive stress experienced by the programmers when developing or maintaining code?

The cause and effect is even more unclear for the code product metrics of coupling and cohesion. Code written in adherence to the test-driven process was seen to be more coupled at times than code written by other processes but there's uncertainty if the coupling is of bad nature at all since program components need to interact somehow. Reports of cohesiveness state, although not definitively, that test-driven programs might have a higher cohesion. But like for complexity, there seems to be a deficiency in the studies that cover coupling and cohesion; these attributes are not included in many studies. Then again, if there are no plausible theories about the actual importance of these attributes, there might be need to consider whether or not it is worth to analyze the attributes further.

When development is test-driven, it means a number of tests will be written in the process. Tests, which will eventually cover a good part—three fourths or over—of the functional source code given that the developers stick to the test-driven process and remember to implement the code through the tests. However, experienced developers might attain similar coverage levels without the use of test-driven development if an iterative, rapid development process is used instead where tests are written after finishing functional sections of code.

High structural coverage can result in greater confidence when changes to the code are needed but the reports of the mutation score indicator show that the confidence might not be that well placed. High statement or branch coverage doesn't always mean that the tests are capable of picking up errors in the code since the results of the mutation scores suggest a similar error detection capability regardless of coverage and the development method used in the process.

Tests carry a weight with them. For every two lines of functional non-test source code, at least a line of test code is needed if a test-driven development process is followed. In some cases, bits and pieces of functional code might be smaller due to the mode of development, driving a simpler design of code classes. But it has to be accepted that the overall size of test-driven programs increases proportionally with the amount of tests; a fact that not many research reports emphasize.

Whilst there are many factors involved, it might be that the amount of test code

has an effect on the effort required to develop programs in a test-driven way. The reports seemed to be quite unanimous that it takes more time to develop programs with test-driven development. Since there is no need for elaborate designs and grand plans, it can be faster to get *started* with a test-driven process as was shown in an experiment reviewed in the survey but the toll for the tests has to be paid in the long run.

A longer time to develop doesn't necessarily mean that the productivity of test-driven teams would be all that much lower—that is, if it makes sense to consider productivity at all due to the impact of other contextual factors. Still, it is not uncommon for productivity to drop when industrial teams are adopting test-driven development and the reduction in speed can be significant at times. Then again, in academic circumstances, students seem to surpass their peers at times when they're subjected to test-driven practices. Experience can explain some of the differences that exist between students and industry experts; the same goes for other attributes besides productivity, too, as reported in a study that compared students to experts. Experienced or not, developers have been found to be more productive when they are writing test code as opposed to writing functional code. So in any case it is unlikely that writing a program two times larger because of the included tests would take twice as long and hinder productivity that much.

Benefits of test-driven development might not be immediately visible to the customer but there is a chance that test-driven developed programs pass more customer level or acceptance tests which communicates an impression of higher quality to the customer as well. When interviewed, customers might not see the quality differences between a program that was developed with test-driven principles and a program developed in some other way.

Besides thinking of customer value, professional developers have insight into the specifics of the software process and the products that are created in the process. A viewpoint that came up in one of the surveys was that developers felt they were able to produce more maintainable code because of test-driven development. The view of the developers was backed up by the considerable reductions in maintenance effort—compared to other similar projects—during the maintenance period that followed the development. This finding is of interest; it is indeed possible that the benefits of test-driven development can truly be reaped in later stages of the product life cycle.

Considering cost, there seems to be a trade-off: increased development effort might lead to increased costs but the reduced maintenance effort can subsequently reduce

costs later. The savings might not realize right away but build up as the project matures. For instance, in a project where development lasted for about eight months, the total effort expended on the project was still somewhat higher after a bit less than nine months of maintenance. In particular circumstances, test-driven development might not even be more expensive as an experiment with students showed; the development method was actually considered relatively cheap.

Inevitably, longer development time is accompanied with a longer time to market for products which could be a factor if time is of the essence. Every software project is different, yet a release date a month later in a six month project provokes thought; can it really be said with confidence that test-driven development caused the delay reported in one of the industry case studies? When the opinion of the developers was asked, they felt that it did.

Test-driven development has advantages that need to be weighed with care against the disadvantages. A shipped product with fewer defects and perhaps a slightly less complex structure are definite advantages. After all, while it seems impossible to rid a product of *all* defects, any unnoticed defect can cause grief later on. Furthermore, it has to be considered a benefit if the application of test-driven development truly creates a better working atmosphere for the developers. In the long term, the increased maintainability is an asset, too. But building quality takes time and there is additional work in creating and reworking the test code. With practice and experience, developers can improve their skills and speed but the tests still need to be written. Disregarding effort, the effect of test-driven development to software development activities and products seems to overall be positive; the development method has wide applicability as long as the limitations are acknowledged case-by-case.

Based on the attributes covered in this literature survey, it is possible to reflect on the future direction of test-driven development research. Already Lehman emphasized the importance of software evolution and stated that software products need to be maintained for extended periods of time [Leh96]. Thus, maintenance and maintainability of software products could be seen important for test-driven development as well. The initial results on maintainability are encouraging but there doesn't seem to be too many studies on maintainability yet to fully confirm the positive effects. Perhaps future research could focus on maintainability and not only on maintenance effort and changeability but on components of evolvable software such as compliance since tools exist to measure some of these attributes.

References

AGD07 Arisholm, E., Gallis, H., Dyba, T. and Sjoberg, D., Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise. *IEEE Transactions on Software Engineering*, 33,2(2007), pages 65–86. URL http://dx.doi.org/10.1109/TSE.2007.17.

AGG07 Allen, E., Gottipati, S. and Govindarajan, R., Measuring Size, Complexity, and Coupling of Hypergraph Abstractions of Software: An Information-theory Approach. *Software Quality Journal*, 15(2007), pages 179–212. URL http://dx.doi.org/10.1007/s11219-006-9010-3.

BBB01 Beck, K., Beedle, M., Bennekum, A. v., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J. and Thomas, D., Manifesto for Agile Software Development, 2001. http://agilemanifesto.org. [18.7.2012]

Bec00 Beck, K., *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Reading (MA), 2000.

Bec03 Beck, K., *Test-Driven Development: By Example*. Addison-Wesley, 2003.

BeG98 Beck, K. and Gamma, E., Test Infected: Programmers Love Writing Tests. *Java Report*, 3,7(1998), pages 37–50.

BeN08 Begel, A. and Nagappan, N., Pair Programming: What's in it for Me? *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '08, New York, NY, USA, 2008, ACM, pages 120–128, URL http://dx.doi.org/10.1145/1414004.1414026.

BEW08 Braught, G., Eby, L. M. and Wahls, T., The Effects of Pair-programming on Individual Programming Skill. *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '08, New York, NY, USA, 2008, ACM, pages 200–204, URL http://dx.doi.org/10.1145/1352135.1352207.

BhN06 Bhat, T. and Nagappan, N., Evaluating the Efficacy of Test-driven Development: Industrial Case Studies. *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, ISESE '06, New York, NY, USA, 2006, ACM, pages 356–363, URL http://dx.doi.org/10.1145/1159733.1159787.

Boe88 Boehm, B. W., A Spiral Model of Software Development and Enhancement. *Computer*, 21,5(1988), pages 61–72.

BWD00 Briand, L. C., Wüst, J., Daly, J. W. and Porter, D. V., Exploring the Relationships Between Design Measures and Software Quality in Object-oriented Systems. *Journal of Systems and Software*, 51,3(2000), pages 245–273. URL http://dx.doi.org/10.1016/S0164-1212(99)00102-8.

CAD10 Chelimsky, D., Astels, D., Dennis, Z., Hellesoy, A., Helmkamp, B. and North, D., *The RSpec Book: Behaviour Driven Development with RSpec, Cucumber and Friends*. Pragmatic Bookshelf, 2010.

CaW11 Carroll, J. and Wang, J., Designing Effective Virtual Organizations as Sociotechnical Systems. *Proceedings of the 44th HICSS Hawaii International Conference on System Sciences*, HICSS-44, January 2011, pages 1–10.

CCA06 Canfora, G., Cimitile, A., Garcia, F., Piattini, M. and Visaggio, C. A., Evaluating Advantages of Test Driven Development: A Controlled Experiment with Professionals. *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, ISESE '06, New York, NY, USA, 2006, ACM, pages 364–371, URL http://dx.doi.org/10.1145/1159733.1159788.

CHH01 Cook, S., He, J. and Harrison, R., Dynamic and Static Views of Software Evolution. *Proceedings of the IEEE International Conference on Software Maintenance*, 2001, pages 592–601, URL http://dx.doi.org/10.1109/ICSM.2001.972776.

ChK94 Chidamber, S. and Kemerer, C., A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 20,6(1994), pages 476–493. URL http://dx.doi.org/10.1109/32.295895.

Cri05 Crispin, L., Using Customer Tests to Drive Development. *Methods & Tools*, 13,2(2005), pages 12–17.

Cri06 Crispin, L., Driving Software Quality: How Test-Driven Development Impacts Software Quality. *IEEE Software*, 23,6(2006), pages 70–71. URL http://dx.doi.org/10.1109/MS.2006.157.

Dij72 Dijkstra, E. W., The Humble Programmer. *Communications of the ACM*, 15(1972), pages 859–866. URL http://dx.doi.org/10.1145/355604.361591.

DJC09 Desai, C., Janzen, D. S. and Clements, J., Implications of Integrating Test-Driven Development Into CS1/CS2 Curricula. *Proceedings of the 40th ACM Technical Symposium on Computer Science Education*, SIGCSE '09, New York, NY, USA, 2009, ACM, pages 148–152, URL http://dx.doi.org/10.1145/1508865.1508921.

DoB11 Dogša, T. and Batič, D., The Effectiveness of Test-Driven Development: An Industrial Case Study. *Software Quality Journal*, 19(2011), pages 643–661. URL http://dx.doi.org/10.1007/s11219-011-9130-2.

FBB99 Fowler, M., Beck, K., Brant, J., Opdyke, W. and Roberts, D., *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Reading (MA), 1999.

FBB12 Ferreira, K. A., Bigonha, M. A., Bigonha, R. S., Mendes, L. F. and Almeida, H. C., Identifying Thresholds for Object-Oriented Software Metrics. *Journal of Systems and Software*, 85,2(2012), pages 244–257. URL http://dx.doi.org/10.1016/j.jss.2011.05.044.

FeP97 Fenton, N. E. and Pfleeger, S. L., *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Company, Boston (MA), 1997.

Gal04 Galin, D., *Software Quality Assurance: From Theory to Implementation*. Pearson/Addison Wesley, Harlow, 2004.

Gar00 Garlan, D., Software Architecture: A Roadmap. *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, New York, NY, USA, 2000, ACM, pages 91–101, URL http://dx.doi.org/10.1145/336512.336537.

GeW03 George, B. and Williams, L., An Initial Investigation of Test Driven Development in Industry. *Proceedings of the 2003 ACM Symposium on Applied Computing*, SAC '03, New York, NY, USA, 2003, ACM, pages 1135–1139, URL http://dx.doi.org/10.1145/952532.952753.

GKS05 Gîrba, T., Kuhn, A., Seeberger, M. and Ducasse, S., How Developers Drive Software Evolution. *Proceedings of the Eighth International Workshop on Principles of Software Evolution*, September 2005, pages 113–122, URL http://dx.doi.org/10.1109/IWPSE.2005.21.

Gla03 Glass, R. L., *Facts and Fallacies of Software Engineering*. Addison-Wesley, Boston (MA), 2003.

GOP08 Gómez, O., Oktaba, H., Piattini, M. and García, F., A systematic review measurement in software engineering: State-of-the-art in measures. In *Software and Data Technologies*, Filipe, J., Shishkov, B. and Helfert, M., editors, volume 10 of *Communications in Computer and Information Science*, Springer Berlin Heidelberg, 2008, pages 165–176, URL http://dx.doi.org/10.1007/978-3-540-70621-2_14.

GSM04 Geras, A., Smith, M. and Miller, J., A Prototype Empirical Evaluation of Test Driven Development. *Proceedings of the 10th International Symposium on Software Metrics*, METRICS 2004, September 2004, pages 405–416.

GuJ07 Gupta, A. and Jalote, P., An Experimental Evaluation of the Effectiveness and Efficiency of the Test Driven Development. *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, ESEM 2007, September 2007, pages 285–294, URL http://dx.doi.org/10.1109/ESEM.2007.41.

Hig01 Highsmith, J., History: The Agile Manifesto, 2001. http://agilemanifesto.org/history.html. [18.7.2012]

HiM96 Hitz, M. and Montazeri, B., Chidamber and Kemerer's Metrics Suite: A Measurement Theory Perspective. *IEEE Transactions on Software Engineering*, 22,4(1996), pages 267–271. URL http://dx.doi.org/10.1109/32.491650.

HuH09 Huang, L. and Holcombe, M., Empirical Investigation Towards the Effectiveness of Test First Programming. *Information and Software Technology*, 51,1(2009), pages 182–194. URL http://dx.doi.org/10.1016/j.infsof.2008.03.007.

Hum95 Humphrey, W., Introducing the Personal Software Process. *Annals of Software Engineering*, 1(1995), pages 311–325. URL http://dx.doi.org/10.1007/BF02249055. 10.1007/BF02249055.

JaM10 Jablokow, K. and Myers, M., Managing Cognitive and Cultural Diversity in Global IT Teams. *Proceedings of the 5th IEEE International Conference on Global Software Engineering*, ICGSE 2010, August 2010, pages 77–86, URL http://dx.doi.org/10.1109/ICGSE.2010.17.

JaS06 Janzen, D. and Saiedian, H., On the Influence of Test-Driven Development on Software Design. *Proceedings of the 19th Conference on Software Engineering Education and Training*, CSEET '06, April 2006, pages 141–148.

JaS08 Janzen, D. S. and Saiedian, H., Does Test-Driven Development Really Improve Software Design Quality? *IEEE Software*, 25,2(2008), pages 77–84.

KBP02 Kaner, C., Bach, J. and Pettichord, B., *Lessons Learned in Software Testing*. Wiley and Sons, New York (NY), 2002.

Kos08 Koskela, L., *Test Driven: Practical TDD and Acceptance TDD for Java Developers*. Manning, 2008.

Lam09 Lamsweerde, A. v., *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 2009.

Leh96 Lehman, M., Laws of Software Evolution Revisited. In *Software Process Technology*, Montangero, C., editor, volume 1149 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, 1996, pages 108–124, URL http://dx.doi.org/10.1007/BFb0017737.

LLP10 Liu, L., Li, T. and Peng, F., Why Requirements Engineering Fails: A Survey Report from China. *Proceedings of the 18th IEEE International*

Requirements Engineering Conference, RE 2010, September 27. - October 1. 2010, pages 317–322, URL http://dx.doi.org/10.1109/RE.2010.45.

Mad10 Madeyski, L., The Impact of Test-First Programming on Branch Coverage and Mutation Score Indicator of Unit Tests: An Experiment. *Information and Software Technology*, 52,2(2010), pages 169–184. URL http://dx.doi.org/10.1016/j.infsof.2009.08.007.

MaS07 Madeyski, L. and Szała, L., The Impact of Test-Driven Development on Software Development Productivity — An Empirical Study. In *Software Process Improvement*, Abrahamsson, P., Baddoo, N., Margaria, T. and Messnarz, R., editors, volume 4764 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, 2007, pages 200–211, URL http://dx.doi.org/10.1007/978-3-540-75381-0_18.

MaW03 Maximilien, E. and Williams, L., Assessing Test-Driven Development at IBM. *Proceedings of the 25th International Conference on Software Engineering*, ICSE 2003, May 2003, pages 564–569.

McC76 McCabe, T., A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2,4(1976), pages 308–320. URL http://dx.doi.org/10.1109/TSE.1976.233837.

MCM11 Misirli, A. T., Çağlayan, B., Miranskyy, A. V., Bener, A. and Ruffolo, N., Different Strokes for Different Folks: A Case Study on Software Metrics for Different Defect Categories. *Proceedings of the 2nd International Workshop on Emerging Trends in Software Metrics*, WETSoM '11, New York, NY, USA, 2011, ACM, pages 45–51, URL http://dx.doi.org/10.1145/1985374.1985386.

MFC05 Middleton, P., Flaxel, A. and Cookson, A., Lean Software Management Case Study: Timberline inc. In *Extreme Programming and Agile Processes in Software Engineering*, Baumeister, H., Marchesi, M. and Holcombe, M., editors, volume 3556 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, 2005, pages 1297–1298, URL http://dx.doi.org/10.1007/11499053_1.

MRW77 McCall, J. A., Richards, P. K. and Walters, G. F., Factors in Software

Quality. Technical report for the Rome Air Development Center (ISIS), General Electric, 1977.

MuC05 Mugridge, R. and Cunningham, W., *Fit for Developing Software: Framework for Integrated Tests*. Prentice Hall, Upper Saddle River (NJ), 2005.

MVM10 Machado, P., Vincenzi, A. and Maldonado, J., Software Testing: An Overview. In *Testing Techniques in Software Engineering*, Borba, P., Cavalcanti, A., Sampaio, A. and Woodcook, J., editors, volume 6153 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, 2010, pages 1–17, URL http://dx.doi.org/10.1007/978-3-642-14335-9_1.

MüH07 Müller, M. and Höfer, A., The Effect of Experience on the Test-Driven Development Process. *Empirical Software Engineering*, 12,6(2007), pages 593–615. URL http://dx.doi.org/10.1007/s10664-007-9048-2.

NaB05 Nagappan, N. and Ball, T., Use of Relative Code Churn Measures to Predict System Defect Density. *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, New York, NY, USA, 2005, ACM, pages 284–292, URL http://dx.doi.org/10.1145/1062455.1062514.

NMB08 Nagappan, N., Maximilien, E., Bhat, T. and Williams, L., Realizing Quality Improvement through Test Driven Development: Results and Experiences of Four Industrial Teams. *Empirical Software Engineering*, 13(2008), pages 289–302. URL http://dx.doi.org/10.1007/s10664-008-9062-z.

NoD03 Nolan, J. and Davies, R., XP @ Connextra, 2003. http://www.agilexp.com/presentations/XPatConnextra.ppt. [25.7.2012]

PaC11 Pančur, M. and Ciglarič, M., Impact of Test-Driven Development on Productivity, Code and Tests: A Controlled Experiment. *Information and Software Technology*, 53,6(2011), pages 557–573. URL http://dx.doi.org/10.1016/j.infsof.2011.02.002.

PaG08 Pacheco, C. and Garcia, I., Stakeholder Identification Methods in Software Requirements: Empirical Findings Derived from a Systematic Review. *Proceedings of the Third International Conference on Software Engineering Advances*, ICSEA '08, October 2008, pages 472–477, URL http://dx.doi.org/10.1109/ICSEA.2008.45.

PeY08 Pezzè, M. and Young, M., *Software Testing and Analysis: Process, Principles and Techniques*. Wiley, Chichester, 2008.

Ram98 Raman, S., Lean Software Development: Is it Feasible? *Proceedings of the 17th AIAA/IEEE/SAE Digital Avionics Systems Conference*, volume 1 of *DASC 1998*, October-November 1998, pages C13/1–C13/8, URL http://dx.doi.org/10.1109/DASC.1998.741480.

Rep04 Reppert, T., Don't Just Break Software. Make Software. *Better Software*, 6,6(2004), pages 18–23.

RuH09 Runeson, P. and Höst, M., Guidelines for Conducting and Reporting Case Study Research in Software Engineering. *Empirical Software Engineering*, 14(2009), pages 131–164. URL http://dx.doi.org/10.1007/s10664-008-9102-8.

RoJ10 Robbins, S. P. and Judge, T. A., *Essentials of Organizational Behavior*. Pearson/Prentice Hall, Upper Saddle River (NJ), 2010.

SGH11 Smit, M., Gergel, B., Hoover, H. and Stroulia, E., Code Convention Adherence in Evolving Software. *Proceedings of the 27th IEEE International Conference on Software Maintenance*, ICSM'11, September 2011, pages 504 –507.

ShC06 Shaw, M. and Clements, P., The Golden Age of Software Architecture. *IEEE Software*, 23,2(2006), pages 31–39. URL http://dx.doi.org/10.1109/MS.2006.58.

ShW08 Shore, J. and Warden, S., *The Art of Agile Development*. O'Reilly, 2008.

Sip06 Sipser, M., *Introduction to the Theory of Computation*. Thomson Course Technology, Boston, 2006.

SMT10 Shull, F., Melnik, G., Turhan, B., Layman, L., Diep, M. and Erdogmus, H., What Do We Know about Test-Driven Development? *IEEE Software*, 27,6(2010), pages 16–19.

Som11 Sommerville, I., *Software Engineering*. Pearson, Boston, ninth edition, 2011.

TCK02 Teasley, S., Covi, L., Krishnan, M. and Olson, J., Rapid Software Development through Team Collocation. *IEEE Transactions on Software Engineering*, 28,7(2002), pages 671–683. URL `http://dx.doi.org/10.1109/TSE.2002.1019481`.

TLS07 Tonini, A., Laurindo, F. and de Spinola, M., An Application of Six Sigma with Lean Production Practices for Identifying Common Causes of Software Process Variability. *Proceedings of the 2007 Portland International Conference on Management of Engineering and Technology*, PICMET '07, August 2007, pages 2482–2490, URL `http://dx.doi.org/10.1109/PICMET.2007.4349584`.

VFS09 Vu, J., Frojd, N., Shenkel-Therolf, C. and Janzen, D., Evaluating Test-Driven Development in an Industry-Sponsored Capstone Project. *Proceedings of the Sixth International Conference on Information Technology: New Generations*, ITNG '09, April 2009, pages 229–234, URL `http://dx.doi.org/10.1109/ITNG.2009.11`.

WaL04 Watt, R. and Leigh-Fellows, D., Acceptance Test Driven Planning. In *Extreme Programming and Agile Methods - XP/Agile Universe 2004*, Zannier, C., Erdogmus, H. and Lindstrom, L., editors, volume 3134 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, 2004, pages 305–328, URL `http://dx.doi.org/10.1007/978-3-540-27777-4_5`.

WMV03 Williams, L., Maximilien, E. and Vouk, M., Test-Driven Development as a Defect-Reduction Practice. *Proceedings of the 14th International Symposium on Software Reliability Engineering*, ISSRE 2003, November 2003, pages 34–45, URL `http://dx.doi.org/10.1109/ISSRE.2003.1251029`.

WNM11 Wilkerson, J., Nunamaker, Jr., J. and Mercer, R., Comparing the Defect Reduction Benefits of Code Inspection and Test-Driven Development.

IEEE Transactions on Software Engineering, PP,99(2011), page 1. URL http://dx.doi.org/10.1109/TSE.2011.46.

Wra10 Wray, S., How Pair Programming Really Works. *IEEE Software*, 27,1(2010), pages 50–55. URL http://dx.doi.org/10.1109/MS.2009.199.

ZBW11 Zhang, M., Baddoo, N., Wernick, P. and Hall, T., Prioritising Refactoring Using Code Bad Smells. *Proceedings of the IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, ICSTW '11, March 2011, pages 458–464, URL http://dx.doi.org/10.1109/ICSTW.2011.69.

ZiN08 Zimmermann, T. and Nagappan, N., Predicting Defects Using Network Analysis on Dependency Graphs. *Proceedings of the 30th International Conference on Software engineering*, ICSE '08, New York, NY, USA, 2008, ACM, pages 531–540, URL http://dx.doi.org/10.1145/1368088.1368161.