Tommi Oinonen

Key Performance Indicators for Automated Testing and Continuous Software Integration

Master's Thesis
Espoo, June 4, 2018

Supervisor: Professor Stavros Tripakis
Advisor: Antti Karjalainen M.Sc. (Tech.), M.Sc. (Econ.)
Evolving software development practices and automation are accelerating the modern software development process. Automation in both software integration and software testing can deliver several widely acknowledged benefits to organizations that invest in it. But in order to reach the full potential of the automation, the process needs to be managed and understood.

Managing any system requires feedback from the managed system and the feedback from software development process is the software development metrics. Traditional software development metrics are concentrating around processes involving people rather than automation. In order to understand the new automated processes, new metrics are needed.

These practices of continuous integration and automated testing, along with the agile development model, no longer produce the artifacts traditionally used to measure and understand the development process. This thesis explores and presents some new software development artifacts and performance indicators derived from them. These artifacts and measures are selected using a top-down goal-oriented process with continuous integration and automatic testing specifically in mind.

These metrics and indicators are developed in a context of embedded systems development at KONE software development. In that context, complex integration flows, tens of thousands of regression test cases and laborious regression analysis involve poorly understood and often informal subprocesses. Nevertheless, these subprocesses offer new measurable software development artifacts that can bring new insight to managing these systems.

**Keywords:** Continuous Integration, CI, Continuous Delivery, CD, Test Automation, TA, Key Performance Indicator, KPI
<table>
<thead>
<tr>
<th>Tekijä:</th>
<th>Tommi Oinonen</th>
</tr>
</thead>
<tbody>
<tr>
<td>Työn nimi:</td>
<td>Suorituskyymittareita testiautomatiolle ja jatkuvalle integraatiolle</td>
</tr>
<tr>
<td>Päiväys:</td>
<td>4. kesäkuuta 2018</td>
</tr>
<tr>
<td>Sivumäärä:</td>
<td>70</td>
</tr>
<tr>
<td>Pääaine:</td>
<td>Tietotekniikka</td>
</tr>
<tr>
<td>Koodi:</td>
<td>T-110</td>
</tr>
<tr>
<td>Valvoja:</td>
<td>Professori Stavros Tripakis</td>
</tr>
<tr>
<td>Ohjaaja:</td>
<td>Diplomi-insinööri, ekonomi Antti Karjalainen</td>
</tr>
</tbody>
</table>


Minkä tahansa järjestelmän hallinta vaatii palautetta hallittavasta järjestelmästä ja ohjelmistokehityksen marrait ovat ohjelmistokehityksen an tamaa palautetta. Pe rinteiset ohjelmistokehitysmitatit keskittyvät prosesseen ilmisten ympärillä automaatioprosessien sijaan. Ymärtääksemme uusia automatisoituja prosesseja tarvitsemme uusia mittareita.

Jatkuva integraatio ja testiautomatio yhdessä ketterien kehitysmallien kanssa eivät enää tuota artefakteja, joita on perinteisesti käytetty kehitysprosessin mit-taamiseen ja ymmärtämiseen. Tästä diplomityö tutkii ja esittelee uusia ohjelmistokehityksen artefakteja ja niistä johdettuja suorituskyymittareita käyttäen tavoiteltaviä top-down prosessia erityisesti jatkuvalta integraatioita ja testiauto maatioita silmällä pitäen.


<table>
<thead>
<tr>
<th>Asiasanat:</th>
<th>jatkuva integraatio, testiautomatio, suorituskyymittari</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kieli:</td>
<td>Englanti</td>
</tr>
</tbody>
</table>
Acknowledgements

I would like to express my gratitude to Antti Karjalainen for his encouragement and feedback and professor Stavros Tripakis for his input throughout the research process. I am especially thankful for their patience and understanding during the final phases of the process.

Secondly, I would like to thank professor Marjo Kauppinen for her guidance in the initial steps and for introducing me to design science research.

Thirdly, I would like to thank Katri Ordning and Antti Hovi for initially proposing the software development metrics as a subject for the thesis along with KONE for the opportunity to conduct this research.

Finally, I want to thank all my colleagues, friends and family who provided their input and encouragement which made this thesis possible.

Espoo, June 4, 2018

Tommi Oinonen
## Abbreviations and Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>KPI</td>
<td>Key Performance Indicator</td>
</tr>
<tr>
<td>CI</td>
<td>Continuous Integration or Continuous Improvement</td>
</tr>
<tr>
<td>CD</td>
<td>Continuous Delivery or Continuous Deployment</td>
</tr>
<tr>
<td>TA</td>
<td>Test Automation</td>
</tr>
<tr>
<td>VCS</td>
<td>Version Control System</td>
</tr>
<tr>
<td>SCM</td>
<td>Source Control Management</td>
</tr>
<tr>
<td>GQM</td>
<td>Goal-Question-Metric process</td>
</tr>
</tbody>
</table>
Contents

Abbreviations and Acronyms 5

1 Introduction 8
   1.1 Motivation ................................. 8
   1.2 Research Questions ....................... 9

2 Research method 11
   2.1 Design Science ............................ 11

3 Background 14
   3.1 Software Development Metrics ............... 14
      3.1.1 Software Process Improvement .......... 14
      3.1.2 Performance Indicators ................. 15
      3.1.3 Key Performance Indicators ............. 16
      3.1.4 Goal-Question-Metric .................... 18
   3.2 Test Automation ............................ 19
      3.2.1 Software Testing ........................ 19
      3.2.2 Automated Testing ....................... 20
      3.2.3 Manual Testing and Regression Tests .... 23
   3.3 Continuous Integration ..................... 24
      3.3.1 Integration Pipeline .................... 25
      3.3.2 Continuous Delivery and Continuous Deployment ... 27

4 KONE context 29
   4.1 Software Development at KONE ............... 29
      4.1.1 Test Automation ........................ 30
      4.1.2 Continuous Integration and Delivery .... 31

5 Metrics 33
   5.1 Measurable Artifacts ....................... 33
   5.2 Automated Testing Performance Indicators .... 35
5.2.1 Goals for Automated Testing .......................... 35
5.2.2 Test Coverage and Reliability .......................... 36
5.2.3 Finding and Fixing Defects .......................... 37
5.2.4 Fixing Failing Tests and Regression Flows .................. 38
5.2.5 Test Efficacy ........................................ 40
5.2.6 Test Result Analysis ..................................... 43
5.2.7 Maintenance of Tests and Environments .................... 46

5.3 Continuous Integration and Delivery Performance Indicators .......................... 48
5.3.1 Goals for Continuous Integration and Delivery .................. 48
5.3.2 Continuous Integration ..................................... 49
5.3.3 Continuous Delivery ...................................... 50
5.3.4 Known Stable Mainline ................................... 52
5.3.5 Rapid Feedback ......................................... 52
5.3.6 Maintainability .......................................... 53
5.3.7 Building Automation and Continuous Delivery Capability .................. 54
5.3.8 Potential and Average Performance .......................... 54

6 Implementation ............................................. 56
6.1 Implementation ............................................. 56
6.1.1 Goal-Question-Metric ...................................... 56
6.1.2 Measurement Artifact Collection .......................... 56
6.1.3 Simple Metrics Service .................................... 57
6.1.4 CIgrid ................................................ 58
6.1.5 Defect Report Metrics .................................... 59
6.1.6 Integration Pipeline Metrics ................................ 61
6.1.7 Test Result Metrics ...................................... 63
6.1.8 Challenges ............................................ 63

7 Discussion .................................................. 65
7.1 Identified Performance Indicators ............................ 65
7.2 Conclusions .............................................. 67
Chapter 1

Introduction

1.1 Motivation

Continuous integration and automated testing are widely considered as an integral part of modern agile software development. The have both been essential tools of agile development from the beginning of the agile movement [12]. Development organizations are transforming their processes to become more agile and they often employ these methods to achieve their goals of delivering higher quality software, faster and more reliably.

The agile development model accelerates the iterations of software development. For the process to be efficient, the overhead of repeating the steps of analysis, design, implementation and testing needs to stay very low. Additionally, if any of these activities can not be shortened, they become bottlenecks for the iterative process. In order to accelerate the development and value creation, each of these steps need to be accelerated together. The development model has progressed from the water fall model to iterative development and continuous development.

The first three of these development activities can be accelerated by changing development practices. In the last decade, many organizations have achieved this by changing their development process and embracing the agile models. But in many contexts, the simple process change has not been able to accelerate the fourth activity of testing.

"They did the requirements on time, the design on time, maybe even the code on time, but testing and integration took much longer than they thought." Agile Manifesto [12]

While the agile software development practices have accelerated the development process, testing is becoming the bottleneck of the processes. Or-
organizations relying on manual testing practices cannot shorten their development cycles since the testers can not keep up with the pace of development. This is especially true for embedded systems development where automated testing becomes nontrivial and continuous integration seems to become an impossible goal [8].

In the agile process, test automation is no longer viewed solely as simply automating manual test cases but as an integral part of the code base of the software. The former view of test automation was linked to situation where the existing, often legacy systems, did not have automated tests written along side their development but afterwards to help maintaining and further developing the systems. The latter, modern view of test automation is that automated tests are written in parallel with the development process. These tests will accumulate and are part of the code base of the system.

Much more has been written about the Return of Investment (ROI) for planned automation than metrics for implemented test automation. These studies on ROI show that investments in both testing and test automation generally are worth the investment [9] [17].

Today, it is mostly agreed that test automation is worth the investment and the interesting question is no longer whether one should automate but rather is the automation effective and how could it be improved. While ROI is a valuable metric for management purposes this thesis focuses on metrics that would help to improve the processes.

1.2 Research Questions

The goal of this thesis is to explore and define useful metrics that would indicate the performance of test automation and continuous integration.

While some of the results are applicable to any domain, the focus of the research is on large software projects with non-trivial testing needs. More specifically, the research is performed in the context of software development at KONE Corporation and one of thesis goals is to produce specific metrics for that organization. The context of software development at KONE is discussed in chapter 4.1.

The main research problem of the thesis aims to answer is:

How to measure the performance of test automation and continuous integration?

This problem is further divided into these three research questions:

**Question 1** When are test automation and continuous integration meeting their goals and contributing to the development process?
Question 2 Which metrics indicate the health and performance of test automation and continuous integration?

Question 3 How can these indicators be generated by automation?

The first question tries to establish some heuristics for continuous integration and automated testing. A lot has been written on how these practices improve the software development process and the quality of produced software in general. This thesis aims to define measurable characteristics of these practices that would allow evaluating their implementations and operating state. More specifically, it is asking the questions what are the properties of an effective and healthy automation process and when is automation useful and worth the investment?

The second question searches for measures of performance that would indicate the health of the automation process and how it is improving the software development effort.

The third question explores whether these measures can be automated and what steps would this automation require. Royce claims in his book [27] that pragmatic software metrics should be automated by-products of the development and integration environment.
Chapter 2

Research method

2.1 Design Science

The goal of this thesis is to produce meaningful measures that would indicate the performance of test automation and continuous integration system.

The research methodology chosen for identifying these indicators is design science. Design science is a research methodology used in information systems research and its objective is to produce a novel artifact that solves an important organizational problem [34]. A design research artifact can be any designed object where a research contribution is embedded in its design [24]. Such artifact can be for example a novel construct, model, method or instantiation of an information system [34].

The artifact is designed using existing knowledge and experimentation. Unlike for example behavioral science, the goal of design science research is utility and not finding the truth. [34]

Design in general is the process of exploring the possible design space and finding a design that best solves a problem. What differentiates design science from routine design is that design science not only produces a solution to the problem but, by doing so, explicitly contributes to the existing knowledge base. [34]

Design science is usually performed in an iterative process that refines the artifacts in cycles. First artifacts are developed using existing knowledge. Then, the new artifacts are evaluated which will produce new knowledge. Next, the new knowledge is communicated and added to the knowledge base and finally by using the new knowledge, the design of the artifacts are redesigned and refined starting the cycle again. [24]

Peffers et al. [24] identified the main activities of design science as:

1. Identify problem and motivate
2. Define objectives of a solution

3. Design and development

4. Demonstration

5. Evaluation

6. Communication

The first activity is to define the research problem and justify the value of the solution. This includes the identification of the domain and context for a solution. The value of the solution is derived from the importance of the problem and the utility of the solution. [24]

Second activity is to define objectives for a solution. These objectives are set to determine the performance of the produced artifacts. The objectives are inferred from the problem identification and they can evolve over the iterations of the process once new knowledge of the problem domain has been accumulated. These objectives can be quantitative or qualitative. [24]

Third activity is creating the design artifact. It could be a new model, method, construct or instantiation. [24]

Fourth activity is to demonstrate how the developed solution is used to solve one or more instances of the problem. Depending on the problem context and the nature of the artifact, this can involve an experiment, simulation, proof, case study, or other appropriate activity. [24]

Fifth activity is evaluation where, the observations and knowledge gained from the demonstration are compared to the objectives. In other words to observe and to measure the performance of the solution with respect to the problem. Knowledge acquired from the evaluation can be used to iterate the design process in the second and third activities. This means the evolution of the objectives of the solution and re-designing the artifact. [24]

The sixth and last activity is communicating the acquired knowledge to the research community and other relevant audiences such as practicing professionals. The acquired knowledge includes products of all the previous activities i.e. the problem and its importance, the artifact, its utility and novelty, the rigor of its design, and its effectiveness. The communication allows for further iterations of the process leading to further refinement of the solutions. [24]

These activities are not necessarily performed in an ordered sequence. The research can be initiated from different steps and there can be varied amount of iteration depending on the circumstances and scope of the research. [24]
CHAPTER 2. RESEARCH METHOD

The research in this thesis is focused on the creation and identification of software development metrics and methods for generating them. Therefore this study is not observational or evaluating former results. Design science is the natural research methodology considering the goals of the thesis and lays a logical structure for the study.

The novel artifacts being designed are the performance metrics and the methods for generating them. The research problem and the goals for the artifacts are defined in the first chapter. The motivation for the research is given in the first and third chapters and the context of the artifact creation is presented in the fourth chapter. The designed artifacts and the reasoning that their design is based on are discussed in the fifth chapter. The designed artifacts are demonstrated and evaluated in the sixth chapter and these results are communicated in the form of this thesis.

Evaluation by instantiation is the natural evaluation method for design science artifacts [25]. The artifacts designed in this thesis are evaluated using the first of the seven evaluation methods defined by Pratt et. al [25] i.e. by a demonstration of the use of the artifact with one or several examples.

Considering the scope of a master’s thesis, only one iteration of design science process is performed in this study and further iterations are left for future research.
Chapter 3

Background

3.1 Software Development Metrics

3.1.1 Software Process Improvement

“At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.” Agile Manifesto [12]

For all software development there is increasing demand for ever better quality software faster, more reliably and with less resources. Therefore development teams and their managers are constantly looking for improvements to their process to increase their productivity.

Continuous improvement is a production management strategy that iteratively improves the production process. Manifestations of this strategy are for example the Japanese kaizen philosophy [23] and lean manufacturing. The idea is to improve the production line by making changes and improvements to the process that in each iteration improve the productivity and the efficiency of the production line.

The idea is to continuously monitor the productivity of the production line and make changes to improve the productivity. These changes are made by first identifying bottlenecks of the process and then trying to remove those bottlenecks.

Viewing software development process as a production line, this strategy can be imported from the management of manufacturing to management of software development. As the trend of software development moves more and more to an iterative process, adoption of the inherently iterative continuous improvement seems a reasonable approach.
CHAPTER 3. BACKGROUND

The key to such improvement processes is to continuously and reliably measure the productivity and efficiency of the production line. In control theory, this is called a closed-loop system. In a closed-loop system, the actions of the controller are determined by some measurement of the systems output or current state. Feedback is needed to make the system controllable and is therefore critical for managing any system. Software development metrics are the feedback for management actions and changes to the process in the context of software development.

3.1.2 Performance Indicators

Each system has several attributes that characterize the system and measurement is an attempt to present those attributes in a quantifiable form. There is an important difference between an attribute and its measurement. Some attributes are impossible quantify in any comprehensive manner.

In order to understand the performance of production process, both the output of the process and the process itself need to be measured. Metrics that characterize the output of the software development process are called software metrics and those that characterize the process are called software development metrics.

In his book *Software Project Management* [27] Royce lists characteristics of good pragmatic software metrics:

- meaningful to manager, customer and performer
- demonstrate a quantifiable correlation between process perturbations and business performance
- objective and unambiguously defined
- display trends
- natural by-product of the process
- supported by automation

Indicators are meaningful metrics that provide insight to the process and its functioning. Such insights can be early warnings of problems and predictions of its state in the future [9].

Indicators might expose a problem but they always need to interpreted. A drop in the number of defects found over time might mean that the software is maturing or that the testing team is on vacation. [27]

In his book [27] Royce lists positive attributes of good indicators:
CHAPTER 3. BACKGROUND

- Easy to interpret and hard to misinterpret
- Collection can be automated
- Non-intrusive
- Consistent throughout the life cycle of the project
- Useful for both management and engineering personnel
- Their fidelity improves across the life cycle of the project

Useful software process management metrics provide insight to waste generated by the process. Scrap and rework are standard measurement perspectives of most manufacturing processes. Such metrics should also recognize the inherently dynamic nature of iterative development processes. Focusing on trends or changes over time is more meaningful for measuring an iterative process. Current trend and current value provide tangible indicators for management action. [27]

The measure of effectiveness for any process is related to the objectives of the process [9]. Performance indicators are a subset of indicators that indicate how well a process is meeting its goals.

Performance metrics can be divided into lagging and leading metrics. Lagging metrics rely on historical data and, in general, represent a longer time frame. They tell what has happened before and respond slowly to changes in the process. Number of defects found in previous month and number of story points implemented in last iteration are examples of lagging software development metrics.

Leading metrics, on the other hand, rely on the current state of the system. Together with a model of the process, they can be used to make projections of the systems future state. Leading indicators are generally less stable and might respond to changes immediately. Number of work items in the backlog or current test coverage are examples of leading software development metrics.

Both leading and lagging indicators are important for management of the system. Only following leading indicators might lead to micromanaging and overcompensating. Relying only on lagging indicators might slow the response time and hide the changes made to the system.

3.1.3 Key Performance Indicators

Key performance indicators (KPI) are a subset of performance indicators. What makes them key indicators is that they are not only tied to the goals
of the process but directly to the goals of the organization. They are selected specifically to encourage behavior and are expected to have transformational impact for the entire organization.

Different parties have interest in different aspects of the project. For example the project managers are interested in the overall project values, testing managers are interested in the status of upcoming releases and development managers are interested in the status of subsystems or components they are responsible for. [27] Even though these different parties have different focuses and scopes of interest, these organizational key performance indicators should be visible, accessible and understandable for the entire organization [33].

According to Weller [35], the key to good project management is to have measurable and countable metrics that can be collected with repeatable process. The process needs to be repeatable in order to produce comparable data over time and different projects. The organization can over time accumulate data on its own projects that can later be used to evaluate the results of the metrics in later projects.

Automation is one approach that will produce repeatable metrics. Royce recommends that the metrics should therefore be an automated by-product of the development and integration environment [27].

Well known phenomenon of measuring people and processes is that you get what you measure. This phenomenon is useful for process improvement
since measuring performance will probably increase performance but it poses
a significant risk when defining such performance indicators. The risk is
that the selected indicators might be incomplete and encourage suboptimal
behavior. To avoid this, the indicators should aggregate information and
individual metrics. [2] In his book, Austin [2] claims that when the available
performance indicators only provide partial information, the organization
would be better off without the sub-optimizing measurements.

Austin also mentions the fact that people do not like to be measured.
Therefore he recommends aggregation of data to hide individual performance.
This produces what he calls information measurements that look at the per-
formance of a system or a process instead of individuals.

While this thesis focuses on objective and quantitative metrics of the
software development process, the subjective metrics of the process are also
valuable to the managers of such processes. It is important to remember that
the process is not mechanical and involves people. The processes are usually
quite complex entities and any data or metrics collected will always portray
a narrow view of the system.

3.1.4 Goal-Question-Metric

The Goal-Question-Metric (GQM) [4] paradigm is a widely used top-down
oriented approach for deriving and selecting metrics. It was originally used
to improve software products and software development processes but since
the underlying concepts are generic, it is applicable in nearly any complex
measurement setting.

The Goal-Driven Software Measurement process is another approach for
creating meaningful metrics for software development and it also promotes
the top-down approach for creating KPIs [21].

Applying the GQM paradigm involves five activities illustrated in figure
3.1.4. Here, these phases are described in a context of measuring a software
development process.

First phase is to identify and select a set of goals for the organization and
the process. Such goals could be customer satisfaction, improved quality or
on-time delivery. [4] The goals can be derived from the organizations strategy
and process model or by interviewing people involved in and responsible for
the process [16].

For each of these goals a set of questions that are quantifiable and as
completely as possible define the goal are generated. Then, a set of metrics
are defined to answer these questions. One metric is often not enough to an-
swer each question and there can be several metrics that help to answer each
CHAPTER 3. BACKGROUND

Define the goals

Generate questions

Specify measures

Develop data collection mechanism

Collect, validate and analyze results

Figure 3.2: Goal-Question-Metric process

question. Conversely, each metric can provide answers to several questions. [4]

Next, mechanisms for data collection are developed. Finally, the data are collected and metrics are validated, analyzed and interpreted. [4]

This process produces a hierarchical structure of metrics that are mapped to questions and further to the original goals. This framework, illustrated in figure 3.3, first in top-down direction defines the relevant metrics and then in bottom-up direction helps to interpret them in the context. [16]

Defining metrics using bottom-up approach is considered problematic and the top-down approach address the issues with defining KPIs as explained in the previous section. When the metrics are derived directly from the organizations goals, only relevant metrics are selected [16]. This also means that the context of the measurements is embedded in the metrics.

3.2 Test Automation

3.2.1 Software Testing

Software testing is a process of determining whether the produced software both does what it should do and does not do what it should not do. Therefore the two main objectives of software testing are to establish confidence and to find defects [9].

The first objective is to gain confidence that the produced software can
perform the operations it needs to and has all the intended features. This confidence arises from the thoroughness of the testing also known as test coverage.

The second objective is about finding issues or flaws in the software. Patton in his book also adds that software testers task is not only to find flaws in the software but support fixing them [22]. This means that testing should also support debugging and provide the developers feedback as valuable as possible.

Earlier finding and feedback on defects is desirable since since most studies agree that the cost of fixing bugs is orders of magnitude more expansive in the later stages of the development [27]. Especially if problems occur after release or in production phase.

In the conventional software development process, integration and testing consumes approximately 40% or more of the life-cycle resources [27]. Numbers such as this were much easier to produce in the context of the traditional software development process where each activity (requirements, design, development, integration and testing) were clearly separated phases. In modern iterative processes, these activities become interwoven. While this has shown to be beneficial to the productivity, it has become much more difficult to measure or assess these activities and their contributions separately.

### 3.2.2 Automated Testing

Test automation and continuous integration are inherent in modern agile development processes. Both test automation and continuous integration are part of the original Extreme Programming practices [5].

In literature, the term *test automation* traditionally refers to the act of converting manual test cases in to automated test cases. This term is increasingly being used to describe automated testing rather than automation
of testing. Automated tests are software developed as part of the software. When the automated test coverage is high, the result can be considered as self-testing code.

Self-testing code is a prerequisite for higher levels of software process automation. A high automated test coverage is needed to assign sufficient trust in the code when it is integrated and even deployed automatically.

Test-Driven Development (TDD) is a development practice that can ensure high test coverage. TDD process is presented in figure 3.4. In TDD, automated tests are written for every new feature and every new feature is tested.

Since automated tests are code, they can have bugs just as the system they are testing, it is important to test both the tests against the system and the system against the tests. The TDD process also confirms that the system initially fails the new test case before the new feature is implemented. This is important because it is easy to write a test cases that do not actually test anything and the system will always pass it. Such tests are useless and only drain testing resources.

Test automation is not limited to writing automated test cases, it is automating everything to do with testing. Neither is it limited to automating test execution to e.g. executing tests during out-of-office times and nights. [3]

Examples of test automation:

- Test execution (test harnesses and test scripts)
- Test generation (test data and script generators)
• System configuration (testing and deployment environments)

• Simulators and emulators

• Oracles (systems determining the expected behavior and outputs)

• Activity recording and coverage analysis (monitoring and test feedback)

• Test management (organizing, prioritizing and metrics)

Automated test are easier to have on new agile projects and to adopt from the beginning. When tests are written in parallel to other development activities, the testability of the system will be considered early enough in the development process avoiding costly redesign later on.

Testability is a nonfunctional feature of a system under tests (SUT) which is vital for any efficient test automation. Testability in most cases means that the internal state of the system under test can be both observable and controllable. Also the quality of the interfaces is an aspect of testability.

Continuing large projects are transforming to more agile methods. Deploying test automation and continuous integration in large projects requires a significant investment both initially and for maintenance.

Automated tests have two costs that are often ignored. The first cost is the significant initial investment that is only paid back if the tests need to be executed repeatably. Once the automated tests are written, the re-execution of the test is relatively cheap.

Second cost is the maintenance of the tests. If the interfaces and the SUT regularly changes after the initial tests are written, the tests are broken and need rewriting which will incur costs. This is often the case with tests for the user interfaces which tend to have less strict interfaces and evolve constantly. Managing test cases and analyzing test failures are also introducing maintenance costs.

Automation tools will nearly always require tailoring. Implementing and maintaining them is quite inevitable [9]. Development of tailored automation tools is known as tools smithing and is happening in nearly every project with higher levels of automation. Even though there are many popular generalized tools for automation of both testing and integration, there is still a need for customization of these tools.

The cost of both the initial investment and maintenance are heavily impacted by the testability of the system. Testability of the system is crucial for automated testing and in many cases the software architecture needs changes to facilitate automation and no amount of external tools will solve this issue [8].
CHAPTER 3. BACKGROUND

![Graph showing accumulated risk of testing lag](image)

Figure 3.5: Accumulated risk of testing lag [13].

Even considering these costs, the investment in automation is widely considered to be worth it. The automation both accelerates the development process and improves quality.

3.2.3 Manual Testing and Regression Tests

Common misconception about test automation is that automating tests simply create savings in testing. As explained in the previous section, test automation in general requires significant investment in both time and resources. The savings only come from the low cost of repeating the tests. This is why automating tests can be considered cost-effective for regression testing. A lot has been written on whether one should invest automating test and the area where test automation is nearly always worth the investment is naturally regression testing.

Over the life-cycle of the project the set of features and functions grows and so will the set of test cases that verify those features. When the system is later updated, it is important to execute all the previous test cases to verify that the changes made did not break a previously verified features.

These test cases validating previous functionality are called regression tests. When the SUT fails a test in the regression test set, i.e. some previously validated behavior or functionality is broken, the result is called a regression.

In order to maintain testing coverage, the regression tests need to be executed whenever the system is updated. Otherwise, the risk of introducing
undetected regressions grows as illustrated in figure 3.5. [13] The value of automated regression tests becomes especially important in large projects with large test sets and long running projects. Where the size of the regression test set grows and manually executing the tests becomes a huge undertaking, manual testing does not scale and therefore test automation is essential for regression testing in any non-trivial project.

Since nearly every test executed against the developed system should be included in the regression set, there is very little reason not to automate tests. There are still several situations where manual testing is needed. Exploratory testing, usability testing, and security testing for example are difficult or even impossible to automate fully.

Manual testing is often a phase in a continuous delivery pipeline. It is often useful for exploratory testing and customer acceptance testing. Although the testing itself is manual, everything else about the phase can be automated, i.e. setting up the environment, notifying all parties and continue to delivery on approval. [6]

### 3.3 Continuous Integration

In traditional software development projects, integrating the developed features in the last stages was a huge and unpredictable operation [27]. A key idea of modern agile development is developing software through short iterations and small manageable changes. When changes are made in small iterations integrating them becomes much simpler and more predictable.

Continuous Integration (CI) is a software engineering technique where all team members who make changes to the code integrate their changes to the common code base regularly. The independent software components are rebuilt and retested as soon as the code has been modified. The end result of this technique is that a common code base is never in a broken state and integration issues should stay small as the individual changes of the developers are constantly integrated. The software should stay buildable, executable and unbroken over these iterations.

This does not mean that the integration is not broken regularly. One purpose of an integration build is to produce feedback for the developers and the failure of integration is rapid feedback to the developer who made the changes [7]. If the changes are integrated regularly the changes stay small and they should be quick to fix. Unintegrated code is essentially hidden and it is invisible to team members and other stakeholders looking at the latest build [19].

CI allows teams to work with a known working stable baseline. This is
CHAPTER 3. BACKGROUND

often hard for teams that are moving to continuous integration especially with an existing code base. First challenge is to get the mainline to be stable and then keeping it stable. [10] A separate pre-CI tests that can be executed manually and informally can help developers to keep the main code base in a stable state but still providing feedback to the developer [8].

Continuous integration emphasizes and enforces a comprehensive build and test automation that additionally improves the quality of the software in general. Each integration is verified by an automated build that includes tests. [10] This requires test automation and self-testing code.

Many companies and organizations claim that they employ continuous integration but very few actually achieve this ideal. Continuous integration is not a homogeneous practice and there are variations in its description in the literature [30]. Nevertheless, each implementation exists in some point in the spectrum between ad hoc manual integrations and completely automated continuous integration.

For example, building software on a timed schedule such as nightly builds is not continuous integration. While it is not exactly continuous integration, an automated nightly build is a fine step on the way when introducing CI. [10] An automated nightly build is still more continuous than some ad hoc manual integration process.

3.3.1 Integration Pipeline

In many contexts CI is used to refer to the tools and technology that automates software building and testing. In this thesis CI refers to the original meaning which is a practice of continuously integrating every new change as soon as its made. There are of course the tools and technologies that facilitate this practice.

It is common to set up a dedicated integration server that will perform integration actions such as compiling the software, deploying software to the test environment, running automated tests and publishing the test results and the compiled software.

Even though the practice of continuous integration essentially does not require any specific technology, it is usually manifests itself as an automated integration pipeline. The pipeline is formed by consecutive integration actions performed by the integration server step by step.

In most of the literature a simple linear pipeline model is presented to explain the continuous integration pipeline. In practice what is considered a continuous integration pipeline and how it is implemented varies considerably in the industry [29]. Stähl and Bosch presented a notation [29] that represents different pipelines as directed acyclic graphs. This model is described in figure
3.6. The nodes represent different pipeline activities and either internal or external triggering factors. Different types of action triggers include [7]:

- On-demand: Manually triggered when needed
- Scheduled: for off-hour processes
- Poll changes: Continuously poll the SCM system for changes
- Event-driven: SCM system triggers the build

The testing pipeline consists of different phases. There are tests that have different features. The balancing act is between latency and bandwidth or in the case of tests between feedback time and test coverage. Unit tests are fast but also less complicated and have a smaller coverage. System tests have larger coverage but are slower.

In the ideal situation tests are fast and the entire pipeline can be run immediately against every commit. In reality this can be either infeasible or require huge resources if the tests can not be executed as quickly as the new commits are produced.

The idea of a pipeline is to achieve parallelism by running different phases of the pipeline in parallel. While slow regression test for the previous commit are still running the pipeline can start running the unit tests against the next commit. [6] This way parallelism is achieved even though each commit will pass each phase in order. If there is need, the latter and slower stages can pool commits and for example run the tests against all commits outside business hours in nightly or weekend run.
The integration system can be viewed as a production line. Similarly to Kaisen in the famous Toyota factory [31], any problem encountered at any stage should halt the entire production line and work will only continue when the problem has been solved. After some time the production line matures and there is rarely reason to stop since most of the issues have been fixed. [31] Integration pipeline should work in similar fashion. The integrity of the integration pipeline should be everyone’s main task when it is broken [7].

Executables under testing are moved to increasingly production-like environments which usually means that the setting up and tearing down of the environment becomes considerably slower. The closer the test environments are to the production environment the more confidence there can be that the software is not broken and the changes were integrated successfully. [14] After passing the final stages the software should be potentially deployable.

3.3.2 Continuous Delivery and Continuous Deployment

Continuous delivery means adding the next logical piece to the continuous integration pipeline. Since the code is always well tested and not broken why not deliver the software to the customer. Where continuous integration refers to regularly integrating the code changes with the common codebase of the development team, continuous delivery implies sharing every verified code change to a customer. The customer in this case could refer to any other stakeholder who is not necessarily a user of the software. The customer can take the latest version of the software in use at any time she chooses. The customer can for example be another team in the same company building a product that is dependent on the delivered software. [14] Another example is an acceptance testing team performing manual testing with the software.

As stated in the Agile Manifesto [12], the ultimate measure of software development performance is the amount of working code with useful features delivered to the customer. When software is actually delivered to customer more often there is also more customer feedback [32]. Continuous delivery helps to maintain less development silos and closer connections between the developers, operations people and the customer [18].

In a study [18] of 15 Finnish IT firms that started to adopt continuous delivery, none implemented it fully. The extent of continuous delivery was more dependent on the company’s operating domain than than the size of the company. Companies in the Web domain had significantly faster deployments than a telecoms company. Different companies had different business goals. Some companies make a business decision to not release as frequently as they could. [18]

Continuous delivery is a technical capability to deliver working software
Figure 3.7: The hierarchy of continuous deployment.

at any time. Continuous delivery does not mean that the delivery necessarily happens after every successful build and packaging of the software. Continuous integration means that you are technically capable to deliver the software continuously to the customer but you make a business decision not to. [14]

The final step in the hierarchy of continuous delivery is continuous deployment. Continuous delivery is often confused with continuous deployment which means that each change is automatically deployed into production [11]. Delivery is not same as releasing. There can be good business reasons to make releases in certain schedule. [12]

Some companies with very advanced testing and deployment automation employ this strategy. They have enough trust in their testing and monitoring that they have made the business decision to take the competitive advantage from the extremely rapid deployments. These companies usually rely on user testing and A/B testing with monitoring to immediately detect any problems and rollback unsuccessful deployments. In the setting of continuous deployment, the ability rollback to the previous version when ever any problems with the deployment occur is therefore vital. The hierarchy of continuous delivery is illustrated in figure 3.7.
Chapter 4

KONE context

4.1 Software Development at KONE

KONE Corporation is a leading manufacturer of elevators, escalators and automated building doors along with other solutions in the area of people flows. KONE is not a regular software company since software is not its main product but software is an increasingly important part of all its products. There is software installed in every machine that KONE produces and it is developed internally at KONE. KONE employs over 55000 people world wide and the software department employs a few of hundred people in Finland and India.

Most of these systems are embedded systems and the products can be considered infrastructure. This means that unlike most software developed today, the life cycle of the products can be decades and updating broken systems could be very costly. The products have very long maintenance and service life cycle.

Most software components have complex interactions between multiple other components that need to function also with legacy components and varying installation setups. For safety and security reasons interactions between some components need to be fail safe. Different laws and regulations in different markets around the world require for example different emergency behavior for the elevators which leads to extensive testing and audit processes.

The development is organized to couple of programs focused in different service areas. Each program consists of several agile component teams responsible for one or more individual software components. The development roughly follows the Scaled Agile Framework (SAFe) [28].

Component teams work in two week iterations that produce an inter-
nal component release. The component teams test in their own automation pipelines with automated tests. Each program has an integration team that handles bundling components together and conducts automated integration testing producing internal releases for the software testing department. The software testing department will conduct both some automated system tests and manual acceptance tests against the system. The software testing department will give the final verdict on the release readiness of the software.

4.1.1 Test Automation

The test automation effort in KONE started in the 2000’s with the aim to reduce the feedback time and accelerate defect fixing. Previously majority of the testing effort was concentrated in the software testing laboratory where they tested releases and test releases. Although the manual testing process was trusted, it occurred too late in the process and issues reported by the testing laboratory were at that stage very expensive to fix. Around the same time the software department started to move to agile development practices and the feedback cycle from the testing laboratory was far too slow for the development iterations. Integration and stabilization of releases had proven to be far too slow and unpredictable.

Today each team writes their own unit tests and functional component level tests. These tests are developed by both developers and testers in the team. Every team is responsible of their own component tests and any automated pipelines. This means that teams can develop automation and practices that best suite their work but on the other hand means that the automation systems can be quite heterogeneous.

Most teams use Robot Framework [26] for functional level test automation and necessary testing libraries and other helping tools are developed to support various testing needs. Also a dedicated test automation toolsmith team supports all teams in testing and integration. The test automation team has developed several in-house tools and extensions to existing tools for test automation purposes. These include simulators and extending the testing framework with libraries tailored to the needs of various projects. Similar in-house development was also found in most of the companies in the industry survey of Kasurinen et al. [15].

KONE has invested considerably in test automation and continuous integration. Much development work has been done to increase the testability of the systems. For some legacy components or parts of components did not have automated tests and the tests have been written afterwards.

A couple of years before this thesis work, another masters thesis study [17] on the return of investment from the test automation estimated au-
automation to be financially beneficial even excluding the achieved benefits of improved software quality. Only looking savings by reducing manual testing and accelerating the testing feedback loop [17].

The test automation employs both software simulations and various hardware simulator setups. Many components are tested in production hardware when available. In other cases and to accelerate testing in some cases all or some of the interacting components can be simulated. Since hardware simulations are slow and require often limited hardware resources, the regression with hardware simulations are not executed after every commit.

Some component teams have thousands of regression test cases and new test cases are written for all new features and functionalities. Managing the test cases and analyzing the test results employs people. The quality of individual test cases can vary considerably. For one component team there are a some tests that are of such a low quality that they regularly break the build even though the features they test are complete and working in all other testing. This had lead to a situation where there were not enough trust in the results of the regression test sets. Identifying and fixing substandard tests has required considerable effort from the testers. Some testing environments are not stable enough for strict regression testing and cannot be used to decide delivery readiness without additional analysis.

4.1.2 Continuous Integration and Delivery

Each component team has considerable autonomy on how they implement their integration pipeline. Different teams deploy varying levels of automation. Most teams have their own automation servers, each of which executes on average approximately 100 unique different integration tasks weekly. Most teams use Jenkins [1] as the continuous integration server. Jenkins is a popular open source automation server.

Kasurinen et al. found out in their study that companies for some reason or another do not employ a completely continuous delivery nor do they aim for that. The actual implementation is on a spectrum. [15] A similar range is present even internally in KONE software development.

One recent obstacle for continuous internal delivery in the company were unstandardized internal delivery mechanisms. This lead some teams not automating the internal release and delivery.

An internally developed software product management tool has helped to standardize the internal delivery practices among different component teams. The automation work is continuing and the automated product software management tool allowed even the software testing laboratory to continuously receive software updates in the beginning of 2018.
The current drive for many teams is to increase the parallelism of the tests execution but that would require even more considerable investments in simulator hardware and virtual test environment development. Virtual testing resources are moving to use cloud resources which should provide more scalability for the virtual testing environments.
Chapter 5

Metrics

5.1 Measurable Artifacts

In his book *Software Project Management* [27] Royce writes about good pragmatic software metrics. He states as one of the key characteristics of such metrics that they should be natural by-product of the process. Such metrics are the least obtrusive and do not produce additional work for the people in the process.

Pragmatic metrics are, in other words, extracted from the various software development artifacts naturally created by the process itself. These artifacts include design documents, requirement artifacts such as user stories, items in a task management system, defect reports, test results, source code, version control commits, and even the produced binaries.

In the traditional software development process, these artifacts were plentiful when the processes were more formal and produced various documentation and reporting artifacts. Today, they are often seen as waste by the proponents of the agile practices.

One clear example of this phenomenon of lacking measurable artifacts are the defect reports. The numbers and types of reported defects over time are few of the most traditional software development metrics used by various software development organizations. They can indicate both the quality and the maturity of the software. When organizations move to comprehensive test automation and continuous delivery, most report a dramatic drop in the defect reports filed [14]. This drop is explained by the fact that thanks to automation the issues are detected so early in the integration that the developers and testers rather fix the issue immediately than bother to write a formal defect report.

For example, this phenomenon was reported in a company called Paddy
Power when they transitioned to continuous delivery [6]. The developers felt that they no longer needed a bug-tracking system after adoption of continuous delivery [6]. Importantly, the developers probably did not make less mistakes developing the software, they were simply caught very early in the process.

While people are creating less of these measurable artifacts, automation produces them as well but those artifacts are often less understood. An example of such artifacts from automation are the automated testing reports. The amount of the testing reports produced by test automation can be huge but understanding how they can be harnessed to similar role that the defect reports play requires some human touch. Harnessing the test results and failure analysis is discussed more latter in this chapter.

The agile practices avoid documentation that is written only for the sake of documentation. While the agile software development practices might produce less of these documentation artifacts, the ones that are produced are even more important and integral to the development process itself.

One important but often missing quality of the measurable artifacts discussed here are the links to other artifacts. Links between artifacts are often needed to measure characteristics of the process that include and span multiple phases. An example of such metric is the lead time of the process i.e. time from conception of the idea to implementation and delivery of the results. In order to estimate this metric there needs to exist some links between the artifacts representing the conception of the idea such as requirements document and then the code and the software where the idea was eventually implemented.

A shared understanding within the organization of the semantics of the measured artifacts is important. When introducing metrics that rely on data collected from for example a task organizing tool, the validity and comparability is highly dependent on the semantics of the tool usage. For example the definition of done and meaning of work in progress. When the tool is used only for organizing and not monitoring the work flow, the user can become very flexible with the semantics at which point the data is no longer completely valid. Differences in practices between different teams and over time might make analysis difficult when comparing data over different time spans.

For example in a case study where attempts of measuring the benefits of introducing continuous delivery were considerably complicated by changes in working practices over the years the data was collected [32]. The developers in that study reported that even though there were guidelines for the practices, they were neither followed nor enforced.

Additional challenges are posed by the cultural factors and attitudes to-
wards these measurable artifacts. Unless, the testers and developers view for example the detected and reported defects as completely natural product of any design process, they can easily leave issues they find unreported and considerably undermine the value of measuring the reports [35].

When the measured artifacts are collected directly from the process, they usually reflect some reality of the process but they can be extremely difficult to interpret unless they are well understood. The top-down model for defining and selecting metrics, discussed in section 3.1.4, is a valuable tool for identifying the measurable artifacts as well. The Goal-Question-Metric approach is used in the following two sections to identify related software development artifacts and further indicators that could be derived from them.

5.2 Automated Testing Performance Indicators

5.2.1 Goals for Automated Testing

Testing goals may vary between different projects and over time. Developers of a safety critical system are more interested in finding every defect than game developers. Developers of new product are more interested in time-to-market while higher quality becomes more important when the market is cornered. [9]

The following is a list of generic goals and associated questions that help to identify relevant performance metrics for automated testing. These goals and identified metrics are discussed in the following subsections.

1. Establish confidence
   
   (a) How wide is the testing coverage?
   
   (b) Are the tests reliable?
   
   (c) Are the test environments reliable?
   
   (d) How mature is the software?

2. Detect more defects earlier
   
   (a) How many defects are found over time?
   
   (b) How many defects slip through testing phases?

3. Test feedback
CHAPTER 5. METRICS

(a) How long is the testing feedback time?
(b) Can detected issues be reproduced?
(c) How quickly can defects be fixed?

4. Maintainable tests

(a) How many tests are there?
(b) How complex is the testing framework?
(c) How much tests need to be reworked?
(d) How much time is spent in result analysis?

5.2.2 Test Coverage and Reliability

When delivering software both the developing organization and customer need to have some confidence that the developed software actually does what it is expected do and performs the required functions. The parties gain confidence in the developed software when it passes tests executed against it. This confidence is, however, highly dependent on the coverage and reliability of the tests.

Test coverage is a proportional measurement of how thoroughly the test cases exercise the system under test. Coverage is the proportion of all tested inputs, usage scenarios and system behaviors out of all possible inputs, usage scenarios and system behaviors. For any realistic system, the testing coverage can never reach 100%.

While the number of test cases is a very bad indicator of test coverage, adding and executing more tests cases against the SUT always improves the coverage even if only by a tiny amount. Nevertheless, the cost of inefficient testing can easily outweigh any benefits of an insignificant increase in coverage.

Test coverage is usually easy to calculate when there is some artifacts the tests can be compared to. These artifacts depend on the level of testing. On low level tests such as unit tests, the coverage can be calculated against the source code. Such coverage is called code coverage. There are different methods of calculating code coverage such as function coverage, statement coverage and branch coverage. These methods check if each function is called, each statement is executed and each path in the control structures is called respectively. Most unit testing frameworks support these metrics and they are easy to automate. [9]

For higher level testing such as functional testing and integration testing, the coverage is calculated against other artifacts such as interfaces and
functions which are usually extracted from the design artifacts. On highest
system level and acceptance tests the artifacts can be requirements, user in-
terface elements or business rules. [9] Automating coverage calculation on
this level requires some linkage between test cases and these artifacts. The
tool automating this would need to know all the artifacts and which test
cases are linked to which artifacts.

The second factor of the confidence the tests are producing is the reliabil-
ity of the tests. When the tests are unreliable, the test results can not be
trusted. Unreliable tests that produce false negative or positive results i.e.
fail even though there is no problem with the SUT or pass even though the
SUT is misbehaving. These are usually bugs in the test cases them selves.

Another type of bad test case is a test that passes or fails based on race
conditions or test execution ordering and is essentially useless [20]. Such test
cases are called flaky tests and are examples of bad tests. Unreliable tests
can not be tolerated in continuous delivery environment where the delivery
is automated and the business decision to deliver is solely based on the test
results.

The reliability of the testing environment also contributes to the over-
all reliability of the tests. An unreliable test environment that occasionally
causes the SUT to fail test cases even when the SUT would in normal condi-
tions pass. Failures due to environmental issues give inconclusive test results.
Such results will hide otherwise detectable defects in the system.

5.2.3 Finding and Fixing Defects

One of the goals of testing is to ensure the quality of the software by ensuring
that the software is as defect free as possible. [22] Hence one of the goals for
automated testing is to detect defects in the SUT. A defect here is defined
very broadly and it refers to any issue in the SUT that should be fixed.

When testers find defects, they are traditionally communicated to the
developers by filing a defect report. These defect reports are usually collected
to some database where their fixing resolution status can be monitored. [22]
These defect reports are possibly the best known and widely used measurable
artifact produced by the software development process. The numbers and
qualities of defects detected during testing are indicators of both the testing
effort and software quality. They can also indicate the maturity of the system.

Not all defects are created equal. They can be everything between super-
ficial style issues to functionality breaking complete failures of the system.
[22] Defect reports are a product of the analysis and judgment of the testers
and meta data associated with them can provide insight to the software under
development.
A well known quality of defects is that the earlier in the process they are found the easier and cheaper they are to fix. [9] The cost rises by orders of magnitude on each step from development phase to acceptance testing to finally production [27].

There are two ways defect fixing can be accelerated by automated testing. Firstly faster feedback i.e. running the tests and getting the results as soon as possible. When developers still remember what they did, it is considerably easier for them to fix the problem. When the feedback comes later, the developer will start implementing a new feature and returning to the previous feature will take time and cause the same effect to the next feature. The second way is to test more often. This will mean that between the executions of the tests there are less and smaller changes. Ideally, the tests are run against every change the developer makes to the code. Isolating the changes from each other will greatly help analyzing the test results. When the changes are smaller, it is much easier to pinpoint what exactly went wrong.

On the other hand if some tests are run against the SUT only every weekend then that week’s development including several features are all interfering and might cause problems in the result analysis. This will cause several problems as defects can hide other defects and some defects will not be detected in that test run. The interference might make the analysis so difficult that the root cause can be left undetected and this would mean that the value of running the tests will decrease.

Test automation happens primarily in the development phase of the of the process. Therefore detecting the defects in test automation is very desirable. Test automation can also happen in other stages of the testing process and test automation itself is usually build in phases. In the ideal situation most defects are found in the early phases and only few are found in the last phases. In order to monitor if this is actually happening defect reports should be produced in every phase.

If all detected defects were collected on every testing level, the fault slip through ratio of each level could be estimated. Then the effectiveness of adding new types of tests or other improvements in each testing phase could be measured. The challenge is that defects detected in the early and automated stages are often not reported due to the effort required. There is therefore need for lighter techniques for defect reporting. One solution for this is discussed later.

5.2.4 Fixing Failing Tests and Regression Flows

The simplest metric used with large test sets is the numbers of passing and failing tests over time. An example of such graph is presented in figure 5.2 A.
The graph depicts the results of a weekly regression test set of one component team over 8 weeks. The graph shows how the overall number of test cases is rising as test cases are moved to this regression set. A few test cases are failing in each iteration and on the fourth week there is a clear regression of few hundred test cases. Most of these regressions are then fixed over the following weeks.

Since these particular regression tests are executed only during weekends, the SUT accumulates approximately 100 new code changes over the working week. There is little isolation for the changes and graph A suggests that after the large regression week only 328 of the 555 failed test were fixed during the following week. Additionally, the SUT fails some number of tests each week and the numbers do not explain if there are test cases that are continuously broken or if the set of failed tests is changing each week.

There are in the simplest terms four interesting scenarios that can be revealed when each test case is followed individually over time. A change in SUT can:

(a) Not affect the test result of previously passing test
   
   * stable development

(b) Break a previously passing test
CHAPTER 5. METRICS

- Unstable development (a regression)

(c) Fix a previously failed test

- Fix a regression

(d) Not affect the test result of a previously failed test

- Inconclusive or fail to fix

These scenarios can be used to divide test cases in four groups. The state diagram for each test case is depicted in figure 5.1.

The graph B in figure 5.2 depicts the same results using these four groups of tests. The top most area represents test cases that are continuously broken and validates the suspicion that there are such test cases in this regression set. Such test cases are poisonous regarding the feedback given by this test set since they might hide and mask other regressions. More alarmingly the number of such test cases is growing over the examined 8 week period. This was probably the result of testers not having time to analyze all failures each week discussed more in section 5.2.6. Other reason could be bad test cases that give inconclusive results after the analysis.

The second area represents tests that are broken that week and the third area represents tests that were fixed that week. On the fourth week 517 test cases were indicating regressions and the following week 464 regressions were fixed while 136 new regressions were added. The bottom area represents the stable test cases that have not been failed in consecutive weeks.

This is just an example of richer and more indicative data of the test results that can be measured when the testing results are collected so that individual test cases can be followed.

5.2.5 Test Efficacy

After automated test cases are written, the most obvious metric to measure is their efficacy is to measure the execution time. Compared to manual testing, the execution times are often greatly reduced. In practice, the time and resources required to benefit from automated tests are mostly spent in other activities.

Savings argued purely on test execution efficiency can be quite misleading when other test activities are not taken into account. The design of automated test cases requires considerable development effort but that one time effort can be repaid when the test is re-executed multiple times. Nevertheless, this is true only when the effort continuously spent on test maintenance,
Figure 5.2: Simple regression test results for one component team and the corresponding regression states chart of the same results of 8 weekend test runs.
setting up environments, tearing down environments and results analysis in addition to the actual execution do not require excessive time or resources.

The efficiency of the tests is strongly related to the quality of the test case and test framework design and implementation. Since the tests themselves are software, they can have similar performance issues as any other software.

The quality of the test is independent from the quality of the implementation that automates it [9]. This means that the same test case can be executed using different tools and the automation can be implemented differently. The inefficiency of an automated test case can lie both in the test case design and the automation implementation.

The testability of the SUT also effects the efficacy of the testing effort. When the testability of the system is low, this needs to be compensated in the design and implementation of the tests. For example, when the internal state of the system can be both observed and controlled, the setup and teardown phases are much more efficient. In addition, the test failure analysis is enhanced when the systems internal state can be observed.

The effort in writing test cases can be measured in the same manner as any other coding work. The collected and measured artifacts could be the items in a task management system or work time reporting tool. This naturally requires that these test developing tasks are somehow separable from other development work. In modern development work the work of developers and testers are often interleaved and performed by the same people.

Test cases are code that should be version controlled along side with the system source code and produced binaries. While various statistical metrics such as number of new test cases and reworked test cases in a week are found in the source code, it would be highly preferable to generate some more tangible artifacts for indicators. Many parties interested in these statistics are not capable to handle source code. Test cases and test suites are usually quite
CHAPTER 5. METRICS

simple entities to collect to a test database where producing different indicators becomes considerably easier than from source code. A test database might also help with creating links between tests and other artifacts. These could be for example links to design artifacts such as requirement documents or defect reports. These type of links would for example allow to estimate test coverage.

Collecting test result artifacts with sufficient meta data allows to estimate the setup, execution and tear down times.

5.2.6 Test Result Analysis

Regression test cases accumulate over the lifetime of a project and when they are automated their execution can be scaled over time. While the automated activities might scale, test case failure analysis does not scale.

In manual testing the analysis of what caused the failure is mostly done in parallel with the execution. When running automated tests, the analysis becomes a post-execution task. [9] Test automation leads to increasingly larger regression test sets and likely more tests to analyze over time.

The analysis time can easily become an issue. Let’s assume 1000 test cases are executed over night. 200 of them fail and the analysis takes approximately 5 minutes per test case. The analysis would take more than two eight hour work days. Even though it is very unlikely that there are 200 separate issues that caused the regressions, it is not certain until all failed cases are analyzed.

The test feedback time is the time from making the changes to receiving the results. It is arguable that the test feedback time ends when the developer receives the results of the analysis of what exactly went wrong even if they had to perform the analysis by themselves.

Some types of testing such as performance tests might not yield simple pass or fail results and will despite automation require some expert analysis. The analysis will easily become a bottle neck rendering the tests useless if they can not provide sufficient feedback to the developers. Such a challenge was identified in one of the cases in the study of Ståhl et al. [29]. The feedback times for the tests can be greatly increased by this situation.

The analysis time can become a limiting factor for the scaling of test automation since it is mostly a manual task. Let’s describe a simple imaginary situation where a single tester has seven work hours daily to spent on writing new test cases and analyzing results from a regression test set that is executed over night. The tester can write a new test case in 30 minutes on average. On average 5% of the test cases fail every night and the tester spends 6 minutes on average analyzing each of those failures. In theory, this would mean that when the project has accumulated 1400 test cases the tester
Figure 5.4: Number of regression tests accumulated to an imaginary project. The project has a single tester writing new test cases and analyzing the results. In the first scenario (blue line) the tester spends 6 minutes on average analyzing each failing test case reaching a theoretical maximum of 1400 test cases. In the second scenario (red line) she spends 12 minutes on average on each failed test case reaching the maximum of 700 test cases.

will spend her entire work day on simply analyzing the results from previous night and no longer has resources to write new test cases. If the analysis time is doubled to 12 minutes per test case, the maximum number of test cases is halved.

These two scenarios are illustrated on a time line of one calendar year in figure 5.4. For the time line it is assumed that the tester only works on weekdays i.e. Monday to Friday. In the first scenario the tester spends half of her working time on simply analyzing the failures by the beginning of May and in the second scenario already in the middle of February.

While this though experiment may seem arbitrary, it illustrates a simplified underlying model of the scaling performance of higher level automated testing with more complicated test cases that require analysis. The chosen parameter values were quite optimistic when compared to the complex testing work performed in a larger component team in KONE where such analysis work is formalized enough to provide some estimates. A tester involved in this analysis work estimated that the analysis takes from two minutes up to more than an hour per test case.

The challenge of the analysis is identifying the exact state or sequence of actions that produced the problematic behavior of the SUT. A failing test result is essentially useless feedback unless the problem can be reproduced.
CHAPTER 5. METRICS

If the analysis work produced measurable artifacts, it would be possible to get estimations of how much time and resources are spent in the analysis work. These analysis artifacts need to be linked to the corresponding test result artifacts.

The test failure analysis results are an interesting artifact that can be used to compensate for the lack of defect reports. As discussed earlier many organizations no longer produce defect reports when the issues are found in automated testing [6].

The same phenomenon is present in the KONE software development to some extend and there are two types of defect reports. First type is defect reports coming from the software testing lab and the second type is defect reports made inside the software department. Defect reports of the first type are results of the mostly manual acceptance test process in the software testing laboratory and include human judgment. Defect reports of the second type are written in the software department and they are often a result of issues detected by other teams or written as a reminder for issues that can not be fixed within the same iteration. Most issues detected with automated testing by the component teams themselves or the integration team do not leave any measurable artifact past the test results.

The test results themselves are not an artifact reliably comparable to defect reports. Some organizations with highly automated test setups automate the defect generation so that a defect report is filed for every failed acceptance test. Defect generated without human judgment can not be used reliably to indicate the maturity or quality of the system under testing. If the test failure analysis produced artifacts that would encapsulate the human judgment of the analyzers, the analysis results could be utilized to produce indicators much how defect reports are used traditionally.

For most component teams in KONE, the failure analysis is an ad hoc activity and does not leave any artifacts. One large multi-site component team performs the analysis in an organized manner in order to ease communication between the two sites. The analysis reports over 9 weeks from this team were manually summarized to produce the numbers presented in table 5.1. The process used by the team allocates test case failures and the analyzed issues into two categories. First category includes issues identified in the SUT which will be reported back to the developers and the second category issues with the test cases them selves. The process can be enriched with additional judgment data such as issue severity and additional categories for environment problems and failures that were inconclusive or left unaanalyzed. A model of this analysis process is presented in figure 5.5. When a test case is failed, the failure can be caused by a defect of either the SUT, the test case or the test environment. Here defect does not mean a defect report needs to
CHAPTER 5. METRICS

<table>
<thead>
<tr>
<th>Week</th>
<th>Real issues in SUT identified</th>
<th>Test cases failed because of issues in SUT</th>
<th>Issues in test cases identified</th>
<th>Test cases failed because of issues in test cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
<td>699</td>
<td>5</td>
<td>50</td>
</tr>
<tr>
<td>2</td>
<td>18</td>
<td>530</td>
<td>7</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>18</td>
<td>625</td>
<td>7</td>
<td>19</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>386</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>14</td>
<td>3715</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>15</td>
<td>1861</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>16</td>
<td>985</td>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>8</td>
<td>14</td>
<td>1345</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>72</td>
<td>5</td>
<td>13</td>
</tr>
</tbody>
</table>

Table 5.1: Result summaries of one component teams test failure analyses over 9 weekend regression test runs consisting of several thousand test cases.

be filed but that at least one of these is components is flawed.

The numbers in table 5.1 show that the number of failed test cases does not correlate with the number of individual issues or defects in the SUT. The table also shows the gradual downward trend in the number of issues in the SUT which indicates that the software is maturing over time.

5.2.7 Maintenance of Tests and Environments

In modern software development, all components of the software testing and deployment are becoming automated and using virtualized environments. This means that both testing and deployment environment are software code just as the developed software. In many instances these two elements are an integral part of the developed software itself. The tests cases are code distributed alongside the software and both the deployment and testing environments are specified using virtual machines or containers described in some specification language. Additionally, test automation regularly requires some custom testing libraries and automation tools that are all software.

The fact that all these components are software means that the same metrics indicating the size and complexity that are used to measure any developed software. These metrics can indicate the maintainability of these systems.

Automated tests are a growing asset which has value. Size of the set does not represent the test coverage but might tell something about the maintenance effort needed.
Similarly to any other software, both the environment and tests are software that will have bugs. These bugs can be found by testing. The environment and the developed software is tested against the test cases and the test cases are tested against the software. The artifacts produced as a by product of test case failure analysis which is described in the previous section can be used to measure issues detected in all of these three components of the system.

Considering embedded systems testing, the hardware is an integral part of the environment of the software. For testing purposes, virtualization and simulation are valuable tools to improve the testability of the system for automation. However, in order to establish sufficient confidence that test results received from the testing environment adequately reflect the systems behavior in the production environment, tests mimicking the production environment as closely as possible are important. Hardware components as part of the testing environment can be unreliable and often can not be scaled easily. Unreliability of hardware is important to detect since it will greatly effect the reliability of the tests. Oftentimes fixing issues with hardware is expansive and some production environments are prohibitively expensive to duplicate.

The maintenance was identified as a major obstacle for test automation by Kasurinen et al. [15] in their industry survey. For example one the companies under that study had scaled down their test automation efforts because the required work to analyze the results from the automated tests
negated the gains compared to manual testing.

5.3 Continuous Integration and Delivery Performance Indicators

The following is a list of generic goals and associated questions that help to identify relevant performance metrics for continuous integration. These goals and identified metrics are discussed in the following subsections.

5.3.1 Goals for Continuous Integration and Delivery

1. Continuous Integration and Delivery [7] [14]
   (a) How frequently developers commit to mainline? [30]
   (b) How often are features delivered?
   (c) How quickly can a simple change be delivered? [10]
   (d) Can software be deployed at will? [20]

2. Known stable mainline [10]
   (a) How quickly are issues in mainline fixed? [10]
   (b) Is fixing broken pipeline prioritized over new development? [10]

3. Reduced risk of release failure [14]
   (a) How reliable are deliveries and releases?
   (b) How production-like are the testing environments?

4. Fast feedback [29]
   (a) Can developer can wait for feedback before moving to next task?
   (b) Are builds failing fast? [7]
   (c) Is progress visible to all parties? [10]
   (d) Is automation capacity en par with traffic it needs to handle? [29]
   (e) Are most problems identified in early stages?

5. Maintainable [29]
   (a) How much effort is spent in maintaining the automation?
   (b) How complex is the integration process?
5.3.2 Continuous Integration

The core of the practice of continuous integration is to create small manageable changes to the system and integrate these changes regularly with the system. Stahl and Bosch in their study [30] of continuous integration literature identified frequent integrations as a common statement. Most sources advocate frequent commits. The difficult question is how frequent. For example Fowler [10] presents as a rule of thumb that each developer should commit at least once a day.

Hence, the key artifacts for measuring continuous integration are the code changes the developers contribute. These changes can usually be tracked in a source control management system (SCM) or a version control system (VCS). The smallest individual item of these changes is often referred to as a commit.

When the practice of continuous integration is aided by the use of an integration server and an integration pipeline, it is valuable to able to link how these code changes pass through the system. The integration pipeline consists of various activities or tasks such as compiling and testing the software. These tasks generate meta data that should preferably be possible to associate with the appropriate code changes that were implemented between the previous execution of the task.

Collecting the meta data artifacts allows to estimate both the frequency and size of changes that are being integrated. There is no consensus in the literature [30] on how frequent the integrations should be but collecting this data allows to follow its trends which can be a valuable indicator.

The size of these changes can also be estimated and there are different metrics for representing the size of a change. Considering individual commits, the size of change can be estimated as the number of source code lines changed or number of source code files changed.

When considering later slower stages of the integration pipeline, the changes are pooled either because some changes failed an inspection in previous stages and the build was terminated or because of not entirely continuous stages such as nightly regression tests which accumulate all the changes from the previous day. In cases where the changes are pooled the size of the accumulated changes can in addition to the previously mentioned metrics be estimated using the number of the commits accumulated.

Again, there is no ideal size for the changes but the trends are valuable indicators. Nevertheless, the changes should be manageable and small enough to be simple to integrate and therefore following correlation between the size of the changes and build failures and integration issues could be used to learn suitable guidelines.
5.3.3 Continuous Delivery

“Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.” Agile Manifesto [12]

Work done in Continuous Delivery is work delivered. When developers declare something to be done and they are ready to deliver the software, all stakeholders can believe that it is done [11]. Therefore, it is sensible to follow how quickly the changes are delivered.

The full integration flow of one component team in KONE is presented in figure 5.6 using the integration flow notation of Ståhl and Bosch [29]. The figure illustrates how the responsibilities of the integration are divided to three different teams. Data from the integration servers can be used to follow how changes are delivered to these different stages.

Here it is important to identify anchor points in the integration flow. Delivery points are stages where software is delivered to the next party. From Component team to integration team and software testing laboratory. One would like to monitor how quickly and how often deliveries are made to

- common mainline of the team (i.e. other members of the team)
- other teams
- integration team
- acceptance testing team

One interesting question is how are these delivery cycles are in sync with each other. Let us consider a situation where the component team can produce deliveries to the integration testing team every day but it takes the integration testing team several days to test each new software version. Then considering the delivery ability of the entire organization, it is much more important to accelerate the integration teams work which is be the bottleneck here.

In a survey [18] of continuous deployment practices the authors employed two deployment cycle time metrics. First is the cycle time to produce internally potentially deployable software. This indicates how fast increments of software are delivered internally but not deployed to production. Second is the actual cycle time to deployment for the users. This indicates how long it actually takes to push software changes to production.

Possibly the most important performance metric considering continuous delivery is the fastest possible time for a code change to propagate to production [18]. This should only consider the “normal” work flow and not any ad hoc patching situations that would skip any part of the testing process.
Figure 5.6: The full integration flow of one component using the integration flow model described in figure 3.6. Auxiliary activities are excluded.
5.3.4 Known Stable Mainline

"Working software is the primary measure of progress." Agile Manifesto [12]

The second most important goal of continuous integration is to always have a stable and working current baseline version of the software available for the developers to build upon. Keeping the integration green and passing the inspections should be the highest priority [10]. When the common baseline where the developers should integrated their changes to is broken, other developers can not integrate their changes and receive reliable feedback on how they succeeded [7].

The target is not to never break the integration but to be able to fix it quickly and not block the other developers work. If the integration never fails the developers do not receive feedback from the integration system. In the ideal situation, the integration fails regularly but thanks to the feedback, the detected problems can be fixed immediately. Keeping track of the integration failures and their reasons is needed to generate indicators of integration stability and fixing.

Few sources [19] [20] propose measuring the fixing times for the integration. Fixing time is the time between the moment the integration was broken until the integration has been proved to be fixed by execution of the pipeline without issues. The time the pipeline is broken is time that other developers can not use the pipeline to get feedback on their changes.

It is also valuable to understand what caused the failure of the integration. Both the results of a root cause analysis and in what stage the failure occurred in should be collected [19]. In the ideal integration pipeline the problems are detected early and the most of the failures should occur in the earlier steps of the integration.

5.3.5 Rapid Feedback

Build performance refers to duration of the build [7]. The more mature the project becomes the more tests and other tasks included in the build accumulate which, while increasing the confidence in the software passing all the inspections, tends to put pressure on the build performance. So even in the project with excellent build performance, the build will at some point degrade its performance. This is why it is very useful to gather metrics on the builds performance over time.

The pressure usually builds gradually over time when new inspections and tests are added. People who build the software in their daily work might be
CHAPTER 5. METRICS

painfully aware when the build times start to degrade but this does not mean
that managers or other people who do not need to wait for the integration
results in their daily work are aware of the situation without shared metrics.

Since CI means frequent commits, every minute saved in build time counts
[10]. Again there is no ideal limit for the build time but the integration should
happen quickly enough that the developer can wait for the feedback before
moving to other tasks. Fowler offers the guideline from Extreme Program-
ming of 10 minutes as a rule of thumb. [10]

Using the meta data from integration build activity and timestamps from
code commits it is possible to define the feedback time as the time from when
the commit was made until the changes have passed an inspection in the
pipeline.

The feedback to developers can be improved by moving the shortest ins-
pections to the beginning of the pipeline. Additionally, the pipeline should
fail as fast as possible. The inspections most likely to fail should be executed
first. [7] Failure statistics of each individual stage of the pipeline can be used
to optimize the build in this way.

5.3.6 Maintainability

Maintainability of the integration pipeline is crucial for both the health of
the pipeline and the success of the project. Modern integration servers al-
low defining the pipelines in a declarative language which means that the
pipelines are software code as well as every other piece in the automation.

The complexity of the integration systems can be illustrated by the figure
5.6 presenting the full but slightly simplified version of the integration flow
of a single software component. In reality, many of the presented activities
are them selves complex and multi-staged and there are dozens of auxiliary
activities that while important for the process are excluded as unnecessary
for the illustration.

To help the maintenance and increase the shared ownership of the inte-
gration implementation, Ståhl et al. advocate sharing the knowledge of the
integration flow in the organization. The flow of changes should be visible
and unambiguous to all. [29] They offer their integration flow model as a
tool for communication.

The maintenance effort required by the continuous integration infrastruc-
ture could also be tracked as separate activity but like testing it is highly
interleaved with all the other development activities. One estimate of the
required maintenance effort was reported to be 7% of the total effort in a
smaller project tracked in a study [19].
5.3.7 Building Automation and Continuous Delivery Capability

For any organization not implementing continuous delivery or automation, the transformation is not going to happen over night. Continuous integration and delivery implementations exist on a spectrum [15]. The changes require significant investments and the benefits which can be hidden in the overall business and performance metrics might go unnoticed if not followed specifically.

For example in a case study [32] of continuous delivery transformation of a company called Solita detected no change in the selected quantitative performance metrics over the inspection period. The qualitative analysis revealed that the project had considerable increase in satisfaction of both the developers and customers and the reliability of the process was improved. The author suspected that the gained benefits were hidden by the fact that the project tripled the number of developers over the analyzed period. Therefore, the author argues that these two factors canceled each other out and that the project could not have been scaled as quickly without continuous delivery. [32]

There are several milestones that could be monitored over time and in different projects. The extend of automation in the integration flow can be followed including the numbers and types of different inspections [7].

In the simplest terms, there are two most important milestones. First is whether the organization uses an automatic chain to potentially deployable software. This would indicates whether code integration and testing are automated. Second is whether the company uses an automatic chain to deployment. This indicates whether code integration, testing, and deployment are automated. [18]

While many organizations claim to implement continuous delivery, few of them actually do. In a survey of continuous deployment, none of the 15 information technology companies implemented full continuous deployment [18].

5.3.8 Potential and Average Performance

Considering the performance of continuous integration or delivery, the aspect of technical ability or potential performance becomes as important as the average performance.

The difference between average performance and potential performance can be explained by comparing a cat and a sloth. Cat is a very agile and fast creature but most of the time it is simply sleeping somewhere looking
as agile as a sloth. Following averaging metrics, the movement performance of the two creatures would seem similar. Nevertheless unlike the sloth, when the cat is motivated and has reason, it can move extremely quickly and reach almost anywhere in an instant.

Regarding the measurement of continuous integration it is arguably relevant to understand both its everyday performance and its ability or potential. Often there are good business reasons not to deliver or deploy every feature when it has been finished even though there are no technical reasons not to deliver. The goal of the process is to reach sufficient level of automation and confidence that would allow any change in the software to be integrated, tested and released at a push of a button [14]. This does not mean that it is always desirable to update the customers software on every new commit which could be several times a day. There are often various sensible business reasons not to integrate everything to everyone constantly but it is still very important to be able do that when needed.
Chapter 6

Implementation

6.1 Implementation

6.1.1 Goal-Question-Metric

One Goal-Question-Metric iteration was used to help define metrics that could be implemented at KONE software development. While this thesis is focused on continuous integration and testing, the metrics implemented in the project where not only limited to these.

The goals were selected from the goals of the program and refined by discussions with the program and testing managers. Some additional goals and current problem areas were identified with discussions with some developers and testers from different teams.

Based on the initial GQM analysis, the most important measurable artifacts and some metrics were identified. The process of defining key performance indicators extended beyond the scope of this thesis and the initial analysis is omitted here but some of the relevant implemented metrics and identified software development artifacts are discussed in the following sections.

6.1.2 Measurement Artifact Collection

Important measurable software development artifacts identified:

- execution meta data from continuous integration servers
- test result files from Robot Framework tests
- work item and defect report data from a Scrum tool
• defect data from the defect management system of software testing laboratory
• meta data from a code review tool
• meta data from the version management system

Some of the identified artifacts are ephemeral in the normal development process while for others the full history of the artifacts is accessible from the data source. Ephemeral artifacts are the integration server meta data and Robot Framework test results. They are lost after few weeks and therefore implementing the collection mechanism for them was deemed a high priority. The other artifacts are accessible from APIs of the source services.

The reason why the test result files are discarded after few weeks is because their size. The amount of logging data they contain means that while they are enabler for the the analysis and debugging work, they can easily reach sizes of gigabytes each. For this reason, a Robot Framework test result archive that serializes the most relevant test result data to a relational database was already used in the software department. Nevertheless, the organization of this database is no longer suitable for the scope of the current testing effort. Most importantly, linking together the results of the same individual test case over time and different test environments is unreliable and sometimes impossible. Hence a new version of the test result archive was implemented. The new database organizes the results so that creating and following these links becomes possible.

The meta data from the integration servers had to be collected as well. A script that uses the API of the Jenkins integration servers to query for the meta data was implemented. The script collects data such as execution timestamps, durations, triggering reasons, commits to the SUT between the previous execution and final status of each execution. These data are serialized to the same database as the Robot Framework test results are added to help create the links between test results and the executions in the integration server.

Ståhl and Bosch in their study [29] called for unobtrusive methods for collecting quantitative data about the integration flows from continuous integration pipelines. The implemented script and database are able to collect such quantitative data using an unobtrusive method.

### 6.1.3 Simple Metrics Service

Most teams had some simple statistics collected from their integration pipelines. In their own view, they had good understanding of their integration flow and
visibility on their data. Managers, other other hand, had very little visibility to the test metrics and had to ask the teams produce the needed metrics on demand.

In order to begin implementing different metrics, simple web based service was implemented. This service had access to the test result and integration server database. It also had access to the data in the defect databases and the scrum tool.

The service allowed quick prototyping of various metrics and visualizations that could be useful for managing the process. The following sections discuss some examples of implemented indicators that were found insightful.

### 6.1.4 CIgrid

Most component teams felt that they had sufficient understanding of their own integration pipeline and its status. Each team had their own metrics and status displays. The teams them selves felt that they had a good status view but the people outside the component teams i.e. integration team and managers were missing the status view and the metrics. The heterogeneity of the monitoring did not help in this case either.

In order to increase the overall status visibility, the continuous integration status of the teams were added to the main page of the metrics service. The status data was organized in a grid that extended the idea of CIgrid [33].

The idea of CIgrid was introduced in 2005 in Nokia Networks. The idea of the grid is to indicate and visualize the automation capability of different projects in the company. The original grid was filled manually using a simple questioner and could be used as a road map for future improvement. In the original grid, its columns represented different projects and rows represented different integration activities and CI practices. The cells would be colored according to the capability of the project in each activity or practice. The grid presents estimates for some important CI metrics. [33]

The grid is designed to both track progress in improving CI capability and share practices between different projects [33]. Moving from no automation to full automation is easier in steps and it is arguably important to monitor the progress. The grid shows which teams projects have less capability and might need more support and on the other hand which teams are the most capable and able to share their expertise. Martin Fowler states that it is much easier to introduce CI or any new technique by finding someone who already employs the technique and see what the the result looks like [10]. In KONE different teams had different levels of capability in different areas and they have great opportunities to learn from each other.

For the metrics service, the CIgrid idea was simplified and extended to
leverage automation to populate it. In the implemented grid, columns represent different CI activities and rows represent components that are grouped by products they are part of. The information in the cells is populated by automation and the only manual task is for the component teams to mark which of their tasks in their integration servers implements each activity in the grid. Teams can mark several task for each cell.

If an automated task on the integration server can be indicted, the cell is colored according to the latest status of that task and other automatically collected statistics such as the latest task duration or test results is shown. If the team can not point to a automation task implementing that activity such as compilation, the cell is colored black denoting that the activity is not automated.

Marking the important tasks on the servers is valuable since each team is every week running on average 100 unique different integration tasks and the status of only dozen are interesting to other teams or managers.

A slightly simplified example of the implemented grid is presented in table 6.1. The grid shows both the general status of CI capability in the organization and the current status of the depicted integration activities.

### 6.1.5 Defect Report Metrics

Defect reports are collected both for defects detected in the software testing laboratory and for defects detected in the software department. The most interesting indicator using the defects from the software testing laboratory is the numbers of defects detected over time and different releases.

This indicator signals the maturity of the software. When the software matures the rate of new reported defects should gradually decrease until it is ready to be released. Sudden changes in this metric can also reflect changes in the testing process. For example if the number of new defects detected drops suddenly, it is more likely that the testing effort has dropped because of tester holidays than that the software has suddenly matured. [22]

An example of this software maturity indicator is presented in figure 6.1.5. the figure shows the monthly number of defects detected in the software testing laboratory for two software releases. This example shows an interesting situation where the number of defects in release A is not falling gradually. The likely reason is that since the testing of the release B started before the release A had matured, the testing resources were suddenly moved away to begin the testing of release B.
<table>
<thead>
<tr>
<th>Product</th>
<th>Component</th>
<th>Compile</th>
<th>Unit tests</th>
<th>Static analysis</th>
<th>Regression</th>
<th>Release</th>
<th>Tests</th>
<th>Release</th>
<th>Testing Lab Automation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product X</td>
<td>Comp A</td>
<td>5 min</td>
<td>6 min</td>
<td>48 min</td>
<td>18 hours</td>
<td></td>
<td></td>
<td></td>
<td>14 hours</td>
</tr>
<tr>
<td></td>
<td>Comp B</td>
<td></td>
<td>14 sec</td>
<td>17 min</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Comp C</td>
<td>7 min</td>
<td>41 sec</td>
<td>31 min</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Product Y</td>
<td>Comp D</td>
<td></td>
<td></td>
<td>31 min</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Comp E</td>
<td></td>
<td>83 sec</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Product Z</td>
<td>Comp F</td>
<td>64 sec</td>
<td>6 min</td>
<td>27 min</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Comp G</td>
<td>2 min</td>
<td>12 min</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Comp H</td>
<td>73 sec</td>
<td>77 sec</td>
<td>13 min</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Cell colors: No automation | Automation passing | Automation unstable | Automation failing

Table 6.1: A simplified example of CIgrid as implemented for the simple metrics service. Times indicated in the cells are execution times and colors indicate whether the task is automated and if so then its latest status.
6.1.6 Integration Pipeline Metrics

The meta data from CI servers allowed the status of the integration tasks and phases to be tracked over time. One implemented meaningful indicator for CI pipelines and individual integration tasks or inspections is the fixing time. Fixing time is the time between the pipeline failing some stage and until the next successful execution. The median fixing time and histogram of fixing times were implemented in the metrics service.

Examples of fixing time histograms are presented in figures 6.1.6 and 6.1.6. The histograms show the overall number and distribution of the lengths of breakages. First histograms present the breakages over a year and the second over the last month. The two figures represent two types of integration tasks. First is a short compilation task executed dozens of times a day and second is a short regression test task executed couple of times a day. While the overall distribution of breakage times in both cases is similar in the previous month compared to the entire year, the longer breakages have not occurred in the last month. Therefore some improvement has happened in the fixing times.

The CI server meta data includes knowledge of the code changes to the software and test cases between consecutive executions of the integration task. This data can be used for two interesting metrics. First metric is for measuring the size of changes that are integrated on each execution. The
Figure 6.2: Fixing time histograms for a 4 minute compilation task.

Figure 6.3: Fixing time histograms for a 2 hour Robot Framework regression test phase.
simple method for measuring the size of changes is the number of commits included in each execution. In the ideal world each commit is integrated one by one. The first rapid stages of the pipeline should always behave in this ideal manner but the latter stages of the pipeline may need to pool the incoming changes. The changes usually pooled for two reasons.

Firstly, the changes are pooled because the software has failed a previous stage of the pipeline and those changes are not integrated any further until the previous change is passing. Second reason are slow pipeline stages such as integration testing that could not otherwise keep up with the incoming changes if the full test set is executed individually against each change. The number of code changes pooled on average in each pipeline stage execution indicates how much difference is in the inspection performance of the pipeline compared to the incoming changes.

The second metric that the change data can be used is to estimate the developer feedback time of each task. The developer feedback time is the time between developer committing the changes and the integration task finishing. The pooling of changes also effects this metric and therefore it is sensible to treat this metric as a distribution of feedback times.

6.1.7 Test Result Metrics

The implemented test result database allows the generation of various test result metrics. The most important advantage of this database is that the links to the pipeline data allow aggregating the results of various distributed test sets. The tests and test sets are constantly moving between specific tasks in the pipelines and acquiring a good over view of the results from e.g. multiple test environments would otherwise be nearly impossible manually. The testing results can be aggregated to CI grid and for any specific subset that the testers or managers need.

The database allows the tracking of individual test cases over time and executions which allows for example the generation of regression flow charts. An example of a regression flow chart is presented in figure 5.2.

6.1.8 Challenges

There were many attempts to further automate some of the metrics discussed in earlier section and attempts to implement some other indicator metric that were deemed infeasible at least in the time scope of this thesis. Some of these implementation challenges encountered during the process are discussed in this section.
CHAPTER 6. IMPLEMENTATION

The issue causing the most challenges is to do with the heterogeneous working practices and tools among the teams. While the freedom to select their own working practices and tools has undoubtedly empowered the teams, it causes many difficulties for implementing automated metrics. For example if the semantics of some measurable artifacts such as defect reports are different among different teams or over time, it is extremely difficult implement generic indicators that would serve more than one team.

Another example is differences in software versioning. When different components employ different version schemes it is nearly impossible to implement automation that has to take in to account any semantic meaning in the version numbers.

Third example is the varying use version control and change management. The interpretation and validity of many indicators that use the commits developers make in the version control system as data are as varying and difficult.

Second challenge is missing links between the artifacts. The best example of a missing link is when the test functional cases can not be reliably linked to any specific requirements or other design artifacts. This situation makes it impossible to estimate testing coverage as there is no way to link test cases to any other artifact which the coverage could be calculated against. If the test cases were tagged with identifiers linking them to specific design artifacts in the beginning of the automation project, maintaining these links would be a feasible task. After the thousands of test cases are already written and the links are missing, the gigantic task to retroactively apply these links is infeasible.

Last example of an encountered challenge was the complexity and size of the integration and testing automation effort. As mentioned earlier, the integration flow diagram of a single component presented in figure 5.6 is greatly condensed and simplified. There were attempts to automatically generate the model of the integration flow using gathered meta data from the integration servers but they were unsuccessful. The integration flows have grown somewhat organically over the years and their true complexity surprised even the people involved in developing themselves.
Chapter 7

Discussion

7.1 Identified Performance Indicators

The goal of this thesis was to identify valuable performance indicators that would specifically aid understanding the performance of test automation and continuous integration. Table 7.1 lists the identified software development artifacts and the indicators that can be derived from them.

The viability and utility of these indicators is validated by their instantiations. All implemented indicators were shown to provide further insight into the processes they are measuring. Their relevance to the goals of the processes they measure is ensured by their selection process. Some of the identified indicators were possible to implement and automate within the scope of the thesis but some were only implemented on as manual prototypes.

While the indicators derived from the regression test failure analysis could not be implemented by automation within the scope of the thesis, the concept was shown to be viable. The plan is to extend the implemented test result database with a tool that would support organizing and recording the results of the analysis work.

The attempts to automatically generate informative models of the integration flow were unsuccessful as explained in section 6.1.8. This topic is nevertheless highly interesting and undoubtedly valuable for any organization with advanced and complex integration automation.

While the context of embedded systems testing and specific issues encountered at KONE software development guided the focus and emphasis of this thesis, the Goal-Question-Metric approach of deriving the metrics employed universal goals for the practices and the results should therefore be mostly applicable in other domains.
<table>
<thead>
<tr>
<th>Source</th>
<th>Artifacts</th>
<th>Indicators</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Defect report database</td>
<td>Defect reports</td>
<td>Defects detected over time (software maturity)</td>
<td>Automated</td>
</tr>
<tr>
<td>Test automation</td>
<td>Test result reports</td>
<td>Regression flows</td>
<td>Automated</td>
</tr>
<tr>
<td>Test result analysis</td>
<td>Regression analysis reports</td>
<td>Test case failure cause statistics</td>
<td>Manual PoC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Lightweight defect reports from automation</td>
<td>Manual PoC</td>
</tr>
<tr>
<td>Integration server</td>
<td>CI server meta data</td>
<td>CIgrid</td>
<td>Automated</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Pipeline fixing times</td>
<td>Automated</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Integration feedback time</td>
<td>Automated</td>
</tr>
<tr>
<td></td>
<td>Integration flow model</td>
<td>Continuous integration and delivery capability</td>
<td>Manual</td>
</tr>
</tbody>
</table>

Table 7.1: The identified software development artifacts and indicators derived from them.
7.2 Conclusions

This thesis is not an exhaustive review of the topic but some viable answers to the main research question were produced. Answers to the first research question about the heuristics of test automation and continuous integration are listed and explored in the fifth chapter. The same chapter also presents several examples of indicators that are searched by the second research question. In the context of the implemented indicators the prospect of automating them is demonstrated which provides answers to the third research question.

The results of this thesis show that there are several practical software development metrics that organizations employing software development automation can take into use. The field of study around continuous integration and automated testing is fairly young. The research into these practices will continue as they will most likely become even more commonplace in every software developing organization. Also the need for metrics to help manage and understand these software development practices will grow in conjunction with the growth of automation. While many organizations today might be able handle their current automation using ad hoc techniques, the future expansion of the automation effort will force them to formalize how it is managed.

As the software development practices are always changing and evolving, the academic research of both new software development artifacts and new software process metrics will be needed in the future. The software development practices of future will both produce completely new artifacts for measuring and render others obsolete.
Bibliography


