EÖTVÖS LORÁND UNIVERSITY

DOCTORAL THESIS

# Quality Aspects of TTCN-3 Based Test Systems


*Author:*
Kristóf SZABADOS

*Supervisor:*
Attila KOVÁCS, Dr. Habil.


*A thesis submitted in fulfillment of the requirements for the degree of Doctor of Philosophy*

*in the*


Eötvös Loránd University
Doctoral School of Informatics
Head: Prof. Dr. Erzsébet Csuhaj-Varjú
Information Systems Program
Head: Prof. Dr. András Benczúr


November 8, 2017

# Declaration of Authorship

I, Kristóf SZABADOS, declare that this thesis titled, "Quality Aspects of TTCN-3 Based Test Systems" and the work presented in it are my own. I confirm that:

- This work was done during the candidature for the degree entirely at Eötvös Loránd University.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.

Signed:

Date:

*"Quality is not an act, it is a habit."* – Aristotle

EÖTVÖS LORÁND UNIVERSITY

# *Abstract*

Doctoral School of Informatics

Doctor of Philosophy

**Quality Aspects of TTCN-3 Based Test Systems**

by Kristóf SZABADOS

Software development is a vital part of everyday life. Software helps in navigating to destinations, communicating with other people, driving the production, distribution and consumption of energy resources. Software drives companies, trades on the markets, takes care of people's health.

Testing these software systems is not trivial. In today's telecommunication world, there are test systems which are comparable in many aspects to the tested systems. Some of these test systems have to support dozens of protocols, simulate millions of unique users, be as robust as the tested systems themselves and provide comparable performance.

The main goal of this thesis is to empirically investigate several different quality aspects of TTCN-3 based test systems in real life settings. *Tests are considered as software products*, and TTCN-3 as a *programming language* used for testing.

In this thesis a list of internal quality attributes applicable to TTCN-3, their connection to international quality standards and an estimation for the real life cost of fixing them will be presented. Empirical investigation revealed that even standardized test systems contain such problems.

Seeing the actual architecture of a test system is important to correctly understand and manage it. This prompted us to create a visualization method, that system architects can use to find architectural issues more efficiently.

Finally, the results of a survey will be presented focusing on better understanding how the knowledge of IT professionals differs having various roles (manager, developer, tester, technical writer), various amount of experience, how they gain new knowledge, how they vary in thinking about their processes and anti-patterns in software development.

All functionality developed for this research is freely available in open source as part of the Titan tool under the name Titanium.

# *Acknowledgements*

I wish to express my sincere appreciation to those who have contributed to this thesis and supported me in one way or the other during this journey.

First of all, I am extremely grateful to my supervisor, Attila Kovács, for his guidance and all the useful discussions and brainstorming sessions. His deep insights in the field of testing helped me at various stages of my research and allowed me to build my knowledge up-to-date.

This research would not have been possible without the help of our industry partners: the Quality Assurance Organization (Test Solutions and Competence Center) of Ericsson Hungary and the Software Technology Organization (DUCN SWT) of Ericsson AB, Sweden. They were kind enough to provide financial support, access to their databases and some of their TTCN-3 source codes. This way we could work on real-life problems and validate our results.

I am grateful to the Quality Assurance Organization of Ericsson Hungary for including the implementations of our algorithms into the open source Titan, making it part of the foundation next generation tests are built upon. The Titan project is accessible as *Eclipse Titan* here: https://projects.eclipse.org/projects/tools.titan

The author would like to thank the Faculty of Informatics of Eötvös Loránd University and the Hungarian Testing Board for supporting this research.

The empirical part of this research would not have been possible without Bernadett Diána Iván processing the fault and review databases at our industry partner and Gábor Jenei, Dániel Poroszkai, Dániel Góbor, Viktor Varga and István Böhm implementing features that were crucial to our investigations

I would also like to thank my friends and coworkers who helped in Code Smell categorization, Technical Debt estimations and reviewing our publications. To users of Titan who showed how important our work was for them and pointed out issues in our implementations.

I would like to thank all those people who offered help distributing and to companies for allowing their employees to fill in our survey (for example Ericsson, Nokia, LogMeIn, NSN, SAP, NNG, Prezi, GE). Our thanks goes also to the meetup groups allowing us to reach their members (Test & Tea, Hungarian C++ Community, Budapest DevOps Meetup, Freelancers in Budapest) and to all visitors of the Hungarian IT professionals group at www.linkedin.com, www.hup.hu and www.prog.hu who filled in our survey.

Special thanks goes to the leaders of the Technical Writers facebook group and Sigma Technologies Hungary, by whom we were able to reach more technical writers.

I would also like to thank my family for supporting me throughout this thesis and my life in general.

# Contents

# Chapter 1

# Introduction

In 1986 the software engineer Barry Boehm observed that the cost of detecting, reporting and correcting defects increases exponentially by the time they are found in the software development process [1]. At that time the overall magnitude of software costs was estimated to be roughly $140 billion per year, worldwide. Yet, until the 2000s tests of these systems were mostly designed, executed and evaluated manually.

Since then the size and complexity of software systems have been growing constantly, together with the quality expectations against these systems. Nowadays the usage of software — developed by 11 million professional software developers [2] — belongs to the everyday life of our society. Software helps in navigating to destinations, communicating with other people, driving the production, distribution and consumption of energy resources. Software drives companies, trades on the markets, takes care of people's health. All of these systems must fulfill very strict (but different) quality restrictions. In the telecommunication area "five nines" (99.999%) availability allows only 5.26 minutes downtime per year — often including planned upgrades and maintenance.

Testing for these expectations is not trivial. Companies producing these systems perform strategically several activities to ensure the required level of quality. They aim at automating tests, managing the size and complexity of tests, that clearly grows with the tested systems. In the telecommunication area this pressure facilitated the ETSI[1] to develop TTCN-3[2], a scripting language used in testing the conformance of communicating systems to standards and for specification of test infrastructure interfaces that glue abstract test scripts with specific communication environments (for a short introduction to TTCN-3 see Appendix A).

The Hello World example (listing 1.1) illustrates the power of TTCN-3. In TTCN-3 it is easy to define an abstract testcase, message based communication ports, to send and receive messages, to describe complex decision making based on timing and matching data to expectations.

The example also demonstrates some problems that can appear in test systems. As part of a large test system this piece of code would be of low quality:

- When the test verdict is not pass there is no information logged, to help debugging.
- The lack of comments is a problem for maintenance. Why does this test exist? Should it be like this?

[1] European Telecommunications Standards Institute
[2] Testing and Test Control Notation - 3

- Unintentionally public definitions can lead to unexpected dependencies.

LISTING 1.1: Hello World in TTCN-3

```
type port PCOType message
{
  inout charstring;
}

type component MTCType
{
  port PCOType MyPCO_PT;
}

testcase tc_HelloWorld() runs on MTCType system MTCType
{
  timer TL_T := 15.0;
  map(mtc:MyPCO_PT, system:MyPCO_PT);
  MyPCO_PT.send("Hello, world!");
  TL_T.start;
  alt {
    [] MyPCO_PT.receive("Hello, TTCN-3!") {
        TL_T.stop;
        setverdict(pass);
       }
    [] TL_T.timeout {
        setverdict(inconc);
       }
    [] MyPCO_PT.receive {
        TL_T.stop;
        setverdict(fail);
       }
  }
}
```

Although tests evolved to be large and complex together with their own standardized language, the internal quality, complexity, structure and evolution of test scripts is not yet a well studied subject. Tassey [3] found that inadequate software testing infrastructure can cost between 22.2 and 59.5 billion USD annually, in the U.S. only.

The TTCN-3 language itself is still under construction and changes rapidly. Since the first version of the standard appeared in 2001 [4] several new features were introduced. Some of them (For example the information hiding techniques in the year 2009 [5]), introduced new keywords breaking previously good codes and turning some previous design techniques into bad practices.

Architects of such test systems had to work for several years, with limited tool support, without being able to see what the actual architecture of their test system looks like. It would be expected, that such systems have architectural issues, but even the visualization of these systems has challenges. Generic graph layout algorithms — freely available to be used to analyze and visualize architecture — don't fit well into daily operations in the industry. Detecting architectural issues or telling what action should be done next is hard (see [6, 7]). In the current situation at the industrial scale most layout algorithms can take several seconds to calculate, as they don't scale well [8], making interactive work impossible for test suites. It is also

not clear who they target: system architects do not have much time to look into the details, and developers lack the high-level view of the systems.

Theoretical and empirical research is also lagging behind. It is unknown whether programming and creating automated tests have similar endeavors, or there is a fundamental difference between them.

## Our contributions

We look at *tests as software products*, we view TTCN-3 as a *programming language*. We analyze software products written in TTCN-3 to see how they compare to "normal" software products. For that reason we extended the open-source tool Titan [9][3]. All functionality developed for our research is freely available as part of the Titan tool under the name Titanium.

In chapter 2 we provide an overview of previous researches related to our work.

In section 3.1 we propose our list of internal quality attributes found applicable for TTCN-3 and we analyse their connections to international standards.

In section 3.3 we show that even publicly available, standardized test systems contain internal quality problems (proving the necessity of our research).

We present our estimations collected from industry experts on the real life cost of fixing the measured quality problems in section 3.4. This enables us to connect the observed internal quality problems, to the amount of work needed to fix them.

In chapter 4 we show that the architecture of the observed test systems (written in TTCN-3) shows phenomenons observed by many as properties of "production" programming languages.

In section 4.2 we present our architecture visualization for these systems. We analyzed all test systems freely available from ETSI's official TTCN-3 homepage[4] for architectural quality and show our results.

In chapter 5 we demonstrate our findings regarding the evolution of a TTCN-3 test system from a software quality point of view. We present historical information on changes in line and project management, development practices, organizational and technical structures, tool support that happened during the five years development period of this system. We show that the internal quality attributes followed predictable patterns during the evolution of this test system.

In chapter 6 we present the results of a survey involving individuals working in software development projects. We show how the mindset of testers and developers is similar, how experience and the size of the company, they work at, affects it.

[3] Titan is a TTCN-3 test toolset used in Ericsson for functional and load testing by more than 4000 internal users.

[4] www.ttcn3.org

# Chapter 2

# Earlier results and related work

Internal software quality and technical debt are tightly linked concepts. *Technical debt* is calculated as the cost of fixing the internal quality problems in an application that, if left unfixed, could put the business at serious risk.

Software systems and test systems accumulate technical debt when during their development short-term goals are traded for long term goals. In a typical development cycle improper documentation, inadequate testing and bug fixing, lack of coordination between teams, legacy code and delayed refactoring, absence of continuous integration and many other factors can lead to increasing technical debt.

In this chapter we overview some of the related aspects: code smells, standards, architectures, software evolution, anti-patterns and the human side of quality.

## 2.1 Technical debt

The term *technical debt* was first used in software development by Cunningham [10] to describe rushing to meet a deadline: "*like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite...*".

Technical debt is a major concern in software development. CAST Consulting estimated [11] the cost of technical debt to be $\$3.61$ per line of code on average. Andy et al. estimated [12] the global amount of IT debt (software and infrastructure debt) to be $\$500$ billion in $2010$. Griffith et al. conducted a study [13] showing that different forms of technical debt can have significant to strong correlation with reusability, understandability, effectiveness and functionality. Ramasubbu et al. investigated [16] the $10$ year long life-cycle of a software package, which had $69$ variants created by customers in parallel. In their empirical investigation they showed that avoiding technical debt resulted in poor customer satisfaction (slow delivery) in the short term, but pays off on the long term with significantly higher software quality and customer satisfaction. Ho et al. proposed an approach [17] which could help product managers to decide the release date of a product.

Yet, technical debt still needs more empirical investigation. Li at el. [18] showed that although the term "technical debt" became widespread, different people use it in different ways, leading to ambiguous interpretations. For example: Holvitie et al. found [14] that in the industry almost $9$ out of $10$ technical debt instances reside in the implementation and that Agile practices are felt by practitioners to reduce or manage technical debt. Mendes et al. [15] found that Agile requirements documentation debt can increase

maintenance effort and project cost by about 47% estimated for the total development phase.

## 2.2 Code smells

*Code smells* were introduced by Fowler [19] as issues in the source code that are not necessarily technically incorrect and do not disable the program from functioning, but might indicate architectural problems or misunderstandings, issues which may correspond to a deeper problem in the system. Since then, Fowler's initial list of 22 code smells has been extensively extended (see e.g. [20, 21, 22]), and code smells have become a metaphor for software design aspects that may cause problems during further development and maintenance of software systems.

Empirical work revealed that parts of the code containing code smells in software systems are changed more frequently than other parts ([23, 24, 25]) increasing the maintenance costs ([25, 26, 27]). Code modified by more developers [28], updated more often [29] or having many changes [30] are more likely to be harder to maintain.

Moser et al. found [31] that in the context of small teams working in volatile domains (e.g. mobile development) correcting smelly code increased software quality, and measurably increased productivity. Zaidman et al. appointed [32] that such corrective actions might result in productivity penalty in the short term.

Zhang et al. [33] provided a systematic literature review on code smells and refactoring strategies based on papers published from 2000 to June 2009. Nearly half of the identified papers (49%) described methods or tools to detect code smells, one-third focused on the interpretation of code smells, and 15% centered on refactoring. There were only a few studies investigating the impact of code smells on maintenance or other quality attributes [23, 34, 35], but none of them were applicable to our test quality smells.

Sjøberg and Yamashita also found in their research [36, 37] that the current level of code smell analysis is only moderately effective at predicting maintenance problems. An extensive review of empirical studies can be found in the doctoral thesis of Yamashita [38].

### 2.2.1 Code smells for testing

In industrial environment automated tests were often "developed" with very little concern for the quality of their code. The quality of tests usually meant code coverage or simply the number of tests written or executed for a product.

While working on a Java project Deursen et al. [39] noticed in 2001 that tests in their test system have their own set of problems and repertoire of solutions, which they translated into code smells and refactorings for the JUnit framework.

In 2007 Zeiss et al. [40] proposed a test specification model derived from ISO 9126 re-interpreting characteristics to be more appropriate in the context of testing. For example the *suitability* is renamed to *test coverage*, as in the context of test specification, the suitability aspect is characterized by the test coverage. They showed the practicability of their smells using their tool

(TRex, see [22, 41]) in the test systems *SIP v4.1.1*, *HiperMan v2.3.1* and *IPv6 v1.1* (available at `www.ttcn3.org`) having altogether 61282 lines of code.

In the present work we take a different way of looking at tests. We look at *tests as software products*. Instead of re-interpreting the quality standards for testing, we re-interpret testing/test development for software product quality. Zeiss et al. [40] chooses TTCN-3 as a *test specification language*. In this work TTCN-3 is viewed as a *programming language*. Software products written in TTCN-3 have to be analysed in order to fulfil quality requirements by applying quality metrics. As we see the two standpoints are not contradictory but rather complementing each other.

We also enable systems being incomplete: we examine software products which are part of larger products (and might contain smaller software products within themselves). In this context test code can be viewed as a product itself. For example, a library of functions that can be used to send and receive messages on a protocol, or a framework that enables the user to do load testing (after configuring the specific messages to be sent/received) can be considered two products. Both examples are software systems that could be used in a standalone mode or as part of a bigger architecture. At the same time both examples share the property of being products that are used by other programmers or test designers. These software products can be delivered to the customer in the form of source codes, enabling further development out of the control of the organization produced them.

## 2.3 Standards

Companies developing complex software systems require quality standards, models and methods to define, perform and institutionalize their quality management processes.

The ISO[1] and IEC[2] published standards 9126 [42] and 25010 [43] embrace the software *product* quality characteristics. Other standards, like ISO/IEC 15504 [44] (SPICE)[3] or CMMI[4] [45], focus on the quality of the software *processes*. GQM[5] [46] describes *measurement* techniques used in software development, while PSP[6] [47] and TSP[7] [48] aims at the human resources and personal processes used during software development.

In the paper of Bánsághi et al. [49] one of the cornerstones was the comparison of the models ISO 9126 and ISO 25010. The article comes to the conclusion that even though the new model is broader, both models suffer from the fact that "different parties with different views of software quality can select different definitions". They state that although both of the standards offer a good frame of reference for software product quality, neither of them offer a practically applicable method for *assessing* quality. The other cornerstone was the fact that there is a wast literature proposing numerous ways of measuring software quality metrics without providing traceable and easily applicable translation to the multi-faceted notation of quality.

[1] International Organization for Standardization
[2] International Electrotechnical Commission
[3] Software Process Improvement and Capability Determination
[4] Capability Maturity Model Integrated
[5] Goal Question Metric
[6] Personal Software Process
[7] Team Software Process

In the software testing area, organizations can use the model based TPINext®[8] [50] or TMMI[9] [51] approaches or the lightweight STEP[10] [52] or CTP[11] [53] to improve their test processes. There exist analytical test improvement strategies as well.

Organizations like ISO, ITU[12] and ETSI have developed the Conformance Testing Methodology and Framework (published as ISO/IEC 9646 [54], ITU-T[13] X.290 to X.296) to ensure that different implementations of a specification conform to the standard they are based upon. TTCN-3 (formerly part 3 of ISO/IEC 9646) is a formal language tailored for testing. A language extended with elements that are commonly required in test descriptions, for example: procedure and message based communication, data matching and concurrent execution. As a standardized language with well-defined semantics TTCN-3 eliminates ambiguities of natural languages. Tests written in TTCN-3 can serve as starting point for any testing activity without platform, testing tool and implementation language dependency. Publicly available and standardized test specifications can significantly improve trust in products, with tests serving as automated and unambiguous requirement specifications providing reproducible and verifiable test results.

## 2.4 Architecture

"The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components and the relationships among them" [55].

Software Design can be 'Wicked' [56]: definitive formulation and stopping rules might not exist, solutions are unique and often 'shades of grey' making it hard to learn and every problem might just be a symptom of an other problem. This 'wickedness' might also mean for architects that each attempt at a solution is a one shot operation: too costly to repeat, and also requiring costly maintenance for the rest of the project's lifetime. It is shown [57, 58, 59, 60, 61, 62] that the reasoning of architects is ad-hoc, not well supported by tools and processes, based on own experiences, prone to bias and fallacy. It is not a far fetched idea that the work of architects should be supported by tools as much as possible.

But, are not. Architecture on this abstraction level is outside the concerns of the TTCN-3 standard, there is no organizing principle defined above modules (such as packages in Java or namespaces in C++). Test systems have grown to large code bases and complex architectures, without much tool support, so they can also be expected to house several issues.

[8] Test Process Improvement
[9] Test Maturity Model Integrated
[10] Systematic Test and Evaluation Process
[11] Critical Testing Processes
[12] International Telecommunication Union
[13] ITU Telecommunication Standardization Sector

### 2.4.1 Architectural smells

One way to look at the architecture is when it's semantic meaning is considered, trying to find code smells.

From the many architectural smells ([63, 64]) in our research we concentrated to the circular dependencies and modules separated from the network, as they are the architectural problems most likely to reflect quality problems [66, 67, 68, 69].

Fiekas et al. [65] discovered that in the investigated systems between $9\%$ and $19\%$ of the dependencies did not conform to the documented architecture. Oyatoyan et al. has shown [66] that components in dependency cycles contain both the majority and the most critical defects. Zimmermann et al. has shown in [67] that binaries of Windows Server 2003 settled down in some dependency cycles had on average two times more failures as other binaries. Schrȯter et al. showed [68] that the actual import dependencies can predict defects. Other empirical studies [69, 70] have shown that even successful and known Java programs contain circular dependencies, and many of these circles forming complex entangled structures. These constructs have a strong impact on the cost of maintenance. They pointed out that "individual developers with no overall view of a system should not be able to reference whatever classes they see fit".

Casierta et al. [71] found that contemporary software visualization techniques need to be more tailored for specific needs to be more widespread. Shahin et al. also found in their survey of $53$ papers [72] that only a few visualization techniques are employed in the industry. Reiss et al. also argued [73] that current visualization is out of touch with the reality of software development. Kuhn et al. found that the embedding of visualization in an IDE [74] provided the participants with immediate estimations of quantity and dispersion.

### 2.4.2 Architecture as a network

From another point of view an architecture can be treated as a network of it's semantic nodes. From this point of view we checked 2 interesting properties of architectures. In a small world network the typical distance $L$ between two randomly chosen nodes is proportional to the logarithm of the number of nodes $N$

$$L \sim logN\,.$$

In a scale-free network the degree distribution follows a power law, that is, the fraction $P(k)$ of nodes having $k$ connections to other nodes is

$$P(k) \sim k^{-\gamma}\,,$$

where $\gamma$ is a parameter typically in the range $2 < \gamma < 3$.

It was shown [75] that scale-free networks have good resistance against random failures, but at the same time have an Achilles' Heel against direct attacks. Vulnerable nodes can be detected using their high number of incoming or outgoing connections.

Several architectural properties of software systems have been shown to be scale-free just like many real-life networks. Scale-free graphs include the physical connection forming the Internet, networks of personal contacts

[76], and even the connectivity graph of neurons in the human brain [77, 78]. It was also shown, that the class, method and package collaboration graphs of the Java language [79] and the object graph (the objects instances created at runtime) of most of the Object Oriented programming languages in general [80, 81] also show scale-free properties.

## 2.5 Evolution

### 2.5.1 Software evolution

Lehman [82] described the evolution of software as the study and management of repeatedly changing software over time for various reasons.

Out of Lehman's *laws of software evolution* [83] the following are the most relevant for our work:

- Law 2: "As an E-type[14] system is evolved its complexity increases unless work is done to maintain or reduce it"
- Law 4: "Unless feedback mechanisms are appropriately adjusted, average effective global activity rate in an evolving E-type system tends to remain constant over product lifetime"
- Law 5: "In general, the incremental growth and long term growth rate of E-type systems tend to decline"
- Law 8: "E-type evolution processes are multi-level, multi-loop, multi-agent feedback systems"

Lehman and Ramil [84], and Lawrence [85] found that commercial systems have a clear linear growth, viewed over a number of releases. Izurieta and Bieman found [86] that Open Source Software products FreeBSD and Linux also appear to grow at similar rates.

Turski showed [87] that the gross growth trends can be predicted by a mean absolute error of order 6%, with

$$S_{i+1} = S_i + \hat{e}/S_i^2 \,,$$

where $S_i$ is the system size at the $i$-th measurement, and $\hat{e}$ can be calculated as $(S_{i-1} - S_1)/(\sum_{k=1}^{i-1} 1/S_k^2)$. Investigating 11 Open Source systems, after removing outliers Ramil et al.[88] could model size trends with $R^2 \geq 0.98$ precision. There is plenty of research ([89, 90, 91, 92, 93]) in which the authors show that the laws seem to be supported by solid evidence.

### 2.5.2 Code smell evolution

The lifespan of code smells was studied by many (see e.g. [94, 95, 96, 97]) to understand software aging better. Chatzigeorgiou et al. found [94] that code smells are usually introduced with new features, accumulating as the project matures, persisting up to the latest examined version. The disappearance of smell instances was usually the side effect of maintenance works, not the result of targeted correcting activities. Peters and Zaidman

[14] Systems actively used and embedded in a real world domain.

concluded [96] that developers might be aware of code smells, but are usually not concerned by their presence. Zaidman et al. [97] witnessed both phased and synchronous co-evolution of tests and production codes.

## 2.6 Anti-patterns

Koenig defined anti-pattern [98] as "just like pattern, except that instead of solution it gives something that looks superficially like a solution, but isn't one". We use this definition to extend the concept of Code Smells to other fields of the industry represented in software development teams. This is necessary to describe the similar phenomena present in the fields of management and technical writing, where source code of any kind might not be directly involved.

Knowledge on anti-patterns in testing is found scattered on the Internet. In their blog posts Carr [99] collected 23 anti-patterns of Test Driven Development and Scott [100] published his observations and ideas regarding anti-patterns. Juristo et al. [101] found that more than half of the existing testing knowledge in 2002 was lacking any formal foundation. Their major conclusion was that the knowledge of testing techniques was very limited at that time. Even the reference book by the International Software Testing Qualifications Board [102] mentions patterns mostly in contexts of testing for a given data pattern, the recognition and handling of anti-patterns is not much covered.

In the field of management Stamelos et al. [103] observed that anti-patterns are likely to appear in student's projects and may cause trouble, affecting the final product. Their introduction to anti-patterns shows why these practices are hard to observe: "In Software Project Management, commonly occurring, repeated bad practices are stated as anti-patterns. These practices are used frequently by software companies because they disguise themselves as an effective and efficient way to resolve common problematic situations, hence rendering it difficult for their negative consequences to be identified. As a result, many project failures in software industry can be attributed to the appearance of anti-patterns...".

In the field of technical writing most books teach techniques using structures and patterns (e.g. [104]). Breaking the pattern of alphabetical ordering, sentence structure or using jargon might be recognized as an anti-pattern. Otherwise, we have not found any article or study aimed at discovering anti-patterns in technical writing.

Femmer et al. found [105] that anti-pattern detection is also helpful in the field of Requirement Engineering, to support quality assurance as a supplement to reviews.

## 2.7 Human side of quality

The role of people in software development is unquestionable: it is people who use the tools, it is people who make the decisions and it is people who apply the changes. To understand the quality aspect of test systems, we must also study the human side of quality.

In their 2013 paper Yamashita et al. [106] conducted a survey on 85 software professionals in order to understand the level of knowledge about

code smells and their perceived usefulness. They found that 32% of the respondents have never heard of code smells nor anti-patterns, only 18% replied to have a strong understanding and to apply this knowledge in his daily activities. Those who were at least somewhat concerned about code smells indicated difficulties with obtaining organizational support and tooling. In their empirical studies [107, 108] they observed that code smells covered only some of the maintainability aspects considered important by developers. They also observed that developers did not take any conscious action to correct bad smells that were found in the code.

Peters and Zaidman concluded [96] that developers might be aware of code smells, but are usually not concerned by their presence. In each system they inspected there were only one or two developers who resolved code smell instances intentionally, or resolved significantly more instances than others (possibly unintentionally).

Calikli et al [109] found similar confirmation bias levels for developers and testers. The size of the company people work for and the amount of experience they had (in years) also had no effect on confirmation bias levels.

The "State of Testing 2015" survey [110] showed that the demand for testers, who can do more than "just testing", is increasing. 81.5% of the testers reported to learn their mastery mostly while doing their work, while only 17% on formal trainings.

The "Worldwide Software Testing Practices Report 2015-2016" [111] survey found that organizations use on the job trainings (72.9%), certifications (51.9%) and formal training (46%) to improve the competency of their employees. This survey also found that Agile management techniques (Scrum, Extreme programming, Kanban) are being adopted often (69.6%) in software development projects.

# Chapter 3

# Quality of test systems – smells, risks, costs

In this chapter we define code smells for TTCN-3, we classify them according to international software quality standards and based on this we show how to measure the internal quality of test systems.

**Thesis 1:** I defined and analyzed TTCN-3 code smells, classified them according to international software quality standards and presented a method for qualifying TTCN-3 based test systems.

**Thesis 2:** I found several internal quality issues in both industrial and standardized TTCN-3 test suites.

**Thesis 3:** I analyzed and assessed the costs of correcting the found internal quality issues of the defined code smell items.

## 3.1 Code smells and categorization

### 3.1.1 Code smell identification

We used a 3 step process to identify TTCN-3 code smells.

First, we have reviewed the databases of source code review documents, errors and problems found in released products, created from year 2006 at our industry partner. These records contained code quality issues which may became show stoppers in any TTCN-3 project's life cycle.

Second, we have also reviewed the rules of PMD [112], FxCop [113], Checkstyle [114], FindBugs [115], xUnit Patterns [116], Martin Fowler's book on refactoring [19] and TRex [117] for static analyzer rules that can be used in testing and in particular for the TTCN-3 language. We found that only a few rules were applicable to our purposes.

Third, we also reviewed the semantic checking and code generation algorithms of Titan for situations which result in low quality or badly performing code.

Based on this work we created the list of code smell rules we found to be applicable to TTCN-3 (see Appendix B).

### 3.1.2 Classification

We classified our code smells during a technical review. The reviewers were experienced, professional TTCN-3 experts from our industry partner. Each rule was discussed and was categorized into the classes which it most likely

belongs to, according to the ISO/IEC 9126 and ISO/IEC 25010 quality models. Most likely means that more than 66% of the review meeting members agreed. In this way there were several rules which fell into multiple categories. For example the rule "infinite loops" belongs to *functionality/suitability* as most likely the program was not intended to operate like that, while it also belongs to the *efficiency/time behaviour* since a program running in an infinite loop is most likely wasting resources. During the review we did not categorize the "FIXME tags" and "TODO tags" rules. The content and severity of this rule depend on the information the developers wished to make visible. As such, each instance may belong to any of the characteristics, completely independently from any other instance. The result of the categorization review can be seen on Figure 3.1 and Figure 3.2.

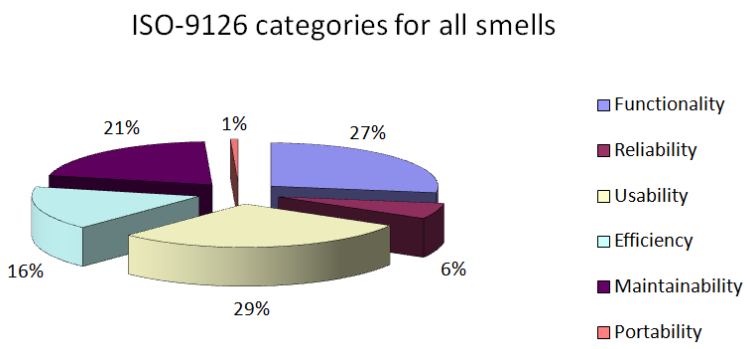



FIGURE 3.1: Code smell classification according to ISO/IEC 9126-1

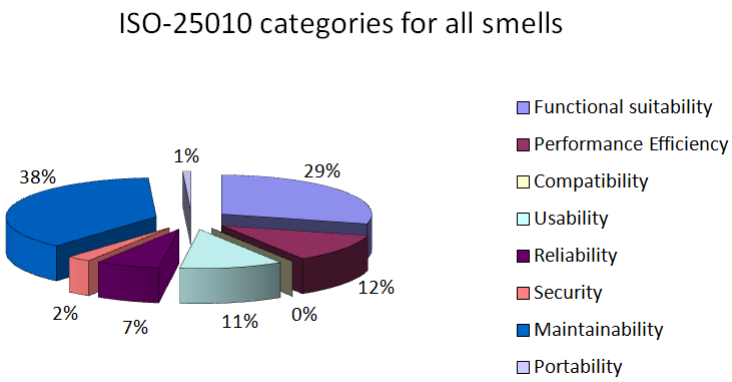



FIGURE 3.2: Code smell classification according to ISO/IEC 25010

## 3.2 The quality risk factor

In order to have an impression about the usefulness of the examined smells we calculated the project risk factors in the usual way:

$$\text{RiskFactor}(\text{proj}) = \sum_{\text{smell}} \text{RelativeOccurrence}(\text{proj}, \text{smell}) \times \text{Impact}(\text{smell}) \,.$$

For the impact estimation we used three classes:

| | |
|---|---|
| 1 | – small impact, |
| 2 | – medium impact, |
| 3 | – large impact. |

There were 4 smells classified into the large-impact class (with ordinal numbers from the smell enumeration): 12, 17, 18, 19; nine smells were classified into the small impact class: 2, 3, 6, 13, 14, 20, 26, 29, 34; all the others belonged to the medium impact category.

In order to determine the classification of the relative occurrences[1] of the smells we used *smell-baselines* on the measured data. For a smell $S$ the smell-baseline $S_b$ means that the smell $S$ is acceptable to occur in every $S_b$ effective lines of code in average. Then, we applied the following categories:

| | |
|---|---|
| 0 | – no smell occurrence, |
| 1 | – rare occurrences ($S_{\text{actual}} > S_b$), |
| 2 | – occasional occurrences ($S_b \geq S_{\text{actual}} > S_b/2$), |
| 3 | – likely occurrences ($S_b/2 \geq S_{\text{actual}} > S_b/8$), |
| 4 | – frequent occurrences ($S_b/8 \geq S_{\text{actual}}$). |

Here $S_{actual}$ means the actually measured relative occurrence in a given project.

Let see an example. Based on the freely available ETSI projects the smell-baseline for the smell *MagicNumber* is 50. In a project $P$ with size 135845 eLOC the actual (measured) value was 5657 occurrences, i.e.,

$$MagicNumber_{\text{actual}} = \frac{135845}{5657} = 24.$$

Hence, this smell occurs more than twice often then the baseline, therefore

$$\text{RelativeOccurrence}(P, MagicNumber) = 3.$$

After calculating the relative occurrences for all smells in project $P$ we were able to determine the risk factor of project $P$. We determined the quality level of the project $P$ by

| | | |
|---|---|---|
| very high | if | $0 < \text{RiskFactor}(P) <= T$, |
| high | if | $T < \text{RiskFactor}(P) \leq 2T$, |
| medium | if | $2T < \text{RiskFactor}(P) \leq 3T$, |
| low | if | $3T < \text{RiskFactor}(P) \leq 4T$, |
| very low | | otherwise. |

The smell-baselines were determined on the basis of the publicly available ETSI projects. We assumed further that the ETSI provided (standardized) projects have good (or very good) quality, i.e., we forced them to fall into the high or very high quality category.

The average value of the risk factors were 60.075, and even the largest risk factor in ETSI projects was below 70 (Figure 3.6). So we selected $T = 35$ as the threshold value.

[1] Here relative occurrence means the size normalized occurrence.

In the time-frame of the research project we were able to implement and measure 35 code smells. Most of the measured code smells are valuable as they point out existing issues. In fact, most of them were present in the examined projects in a large quantity.

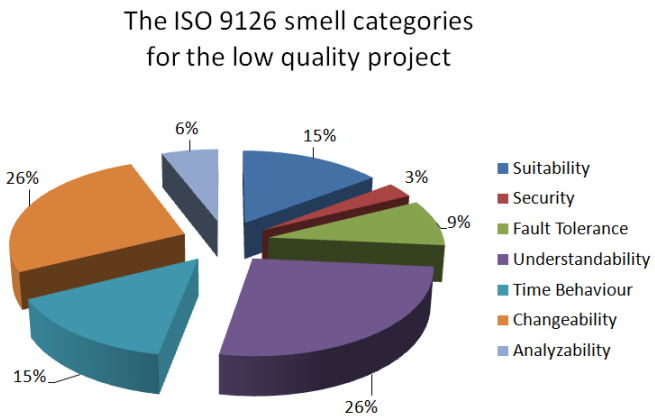



FIGURE 3.3: The most occurred code smells for a low quality (industrial) project categorized according to ISO/IEC 9126-1

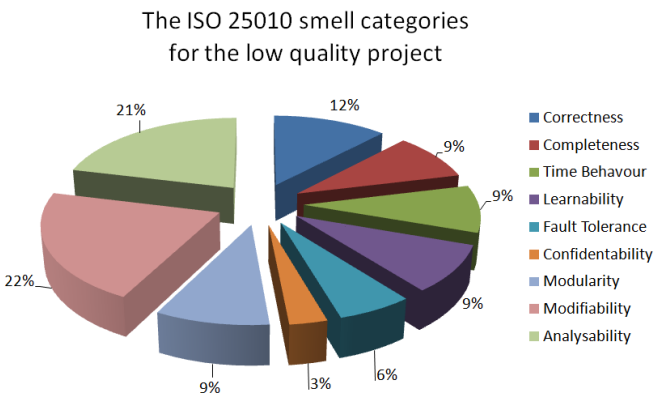



FIGURE 3.4: The most occurred code smells for a low quality (industrial) project categorized according to ISO/IEC 25010

Figure 3.3 and 3.4 shows the code smell penetration for a low quality project at our industry partner, according to the ISO 9126 and ISO 25010 models.

## 3.3 Validation via standardized test suites

### 3.3.1 The analysed projects

We analyzed all test systems which were available at www.ttcn-3.org in January 2014. The webpage lists links to test suites provided by 2 different standardization organizations: ETSI and 3GPP[2]. The projects provided by ETSI were:

[2] 3rd Generation Partnership Project

- WiMax (802.16) Test Suites
- ePassport Readers Interoperability Test Suite
- Session Initiation Protocol (SIP) Test Suite
- IP Multimedia Subsystem (IMS) Test Suites
- IPv6 Test Suites
- Digital Private Mobile Radio (dPMR) Test Suite
- Digital Mobile Radio (DMR) Test Suite
- Intelligent Transport Systems (ITS) Test Suites

The projects provided by 3GPP were:

- 3GPP EUTRA (LTE/EPC) UE Test Suites
- 3GPP IMS UE Test Suites
- 3GPP UTRA UE Test Suites
- 3GPP UE Positioning Test Suites

Most test suites had several parts and some even several versions. We decided to measure all software packages, which were available and contained all source files needed to be able to analyze the project. We measured 40 different packages of test suites.

### 3.3.2 Low level findings

We have identified 32 different kinds of syntactical and semantic issues in the examined projects. We note that only ETSI projects contained syntactical errors. None of the 3GPP projects analysed contained such low level issues.

#### Syntactic issues

To our surprise we found syntactical errors in ETSI testsuites, even though ETSI is the developer of the TTCN-3 language and these freely available software packages most probably have promotional purposes.

An example for this situation is related to how the brackets of formal parameter lists can be used. According to the TTCN-3 standard [5]: if a "template" structure has no formal parameters, the brackets are not allowed to be written out. The BNF dictates:[3]

```
BaseTemplate ::= (Type | Signature)
    TemplateIdentifier ["(" TemplateFormalParList ")"]
TemplateFormalParList ::= TemplateFormalPar
    {"," TemplateFormalPar}
```

In the available projects we have found cases where these empty formal parameter list brackets were present. An example code is:[4]

[3] TTCN-3 standard [5]: Section A.1.6.1.3

[4] Digital Mobile Radio (DMR) Test Suite; in file DMR_Templates.ttcn lines 16

```
template ServiceOpt m_serviceOptDefault () := {
emergency := c_emergencyNone,
privacy := c_privacyZero, ...
}
```

On the other hand, as this kind of notation may also make sense, we can imagine that some tool vendor supports it.

### Semantic issues

To continue our analysis we temporarily fixed the syntactic problems in our lab environment and analyzed the code semantically. This analysis also brought up several issues:

- In some cases we have found assignments in wrong order. For example in the following code the first field of the structure was filled out 3 times [5].

```
template NbrAdvOptions m_nbrAdvOpt_macTlla3
  (template Oct6to15p_macTlla) := {
  tqtLinkLayerAddr := m_macTlla(p_macTlla),
  tqtLinkLayerAddr := m_macTlla(p_macTlla),
  tqtLinkLayerAddr := m_macTlla(p_macTlla),
  otherOption := omit
}
```

- We also found cases of sub-type restriction violations [6].

```
Bit3 ::= BIT STRING (SIZE(3))
...
const Bit3 c_ackNone := '0'B;
const Bit3 c_ack := '1'B;
```

- We found illegal characters in conversion operations that would drive the test to Dynamic Testcase Error at first execution [7].

```
str2oct("SCOP/1.0");
```

- One of the project sets even has an extension of importing from a proprietary file format [8]. This way the test suite can only be used with one vendor's tool.

### Validation

We have contacted ETSI in order to provide us with information on why we could find so many issues in the publicly available testsuites. They were

[5] IPv6 Test Suites; TS 102 351 Methodology and Framework; in file LibIpv2_Rfc2461NeighborDiscovery_Templates_PK.ttcn in line 442

[6] Digital Mobile Radio (DMR) Test Suite; type is defined in file CommonLibDataStringTypes.asn line 30; constants in file DMR_Values.ttcn lines 254-255

[7] IPv6 Test Suites; IPv6 Mobility; TS 102 596 version 1.1.1; in file EtsiLibrary/LibScop/LibScop_Codec.ttcn in line 29; fixed in version 1.2.0

[8] IP Multimedia Subsystem (IMS) Test Suites; Netowkr Integration Testing between SIP and ISDN/PSTN; Part4; in file LibSip/LibSip_XMLTypes.ttcn in line 32

kind enough to direct us to the validation manual ([118]) used by ETSI. Section B.2 of this document describes the validation levels that ETSI uses for its products:

1. Basic: The test suite had been compiled on at least one TTCN-3 tool. Executing the test is not required.

2. Strong: The test suite had been compiled on at least one TTCN-3 tool and executed against at least one SUT (System Under Test). Running to completion is not required and traces might not be analyzed.

3. Rigorous: The test suite must be compiled on more than one TTCN-3 tool and executed on several test platforms. The complete test suite is executed against SUTs from different suppliers. The operation and output of the tests have been validated.

According to this information our findings show that the publicly available test suites were not validated on level 3.

We tried to check this information but we could not find any clear reference. We found that (1) the project web-pages do not list this information, (2) the documents attached to these projects contain only formal descriptions (naming conventions, architectural descriptions, etc.), and (3) most of the packages, containing the source codes, have no non-source code files at all.

On the other hand it was mentioned that the Technical Committee of any given Test Suite has the responsibility to decide which validation level to use. This can result in high diversity in quality among the Test Suites.

### 3.3.3 Measurements

We used code smells (defined in section 3.1) to measure the software quality of test suites.

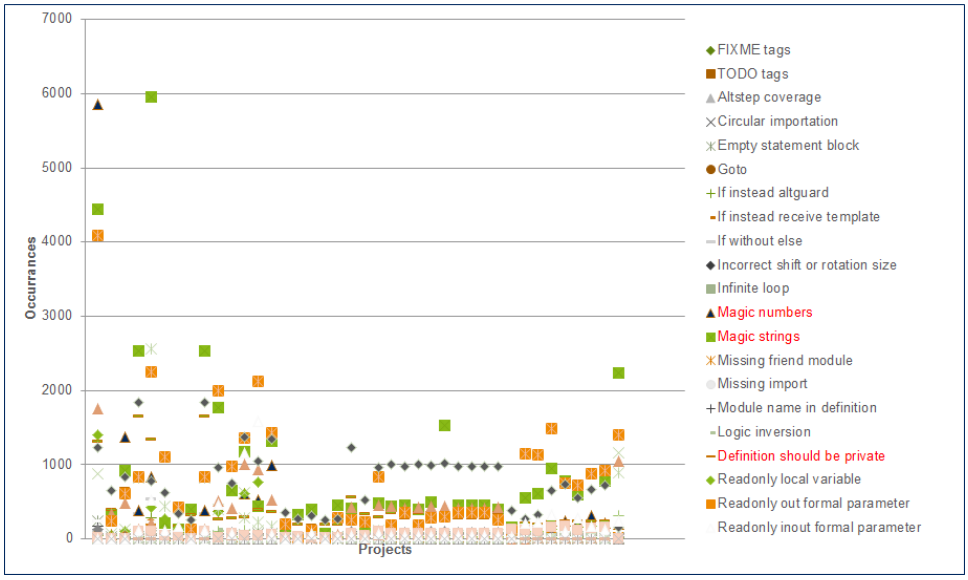



FIGURE 3.5: Code smells measured on the projects (the horizontal axis represents the projects, the vertical axis shows the absolute number of instances found)

Although the amount of code smells we found were different in each project, the frequency of the smells were relatively the same (Figure 3.5).

The top 4 code smells occurred mostly in the examined project were:

- Magic strings and numbers,
- Un-initialized local variables,
- Unused global definitions,
- Definitions that could be private, but are not set so.

Some of these come from the original idea behind the language: let writing testcases be as easy and as fast as possible.

TTCN-3 supports a compact representation of values, enabling high development speed. This also helps burning "magical" values into the source code, which can lead to understandability and changeability problems.

Un-initialized local variables might point out implementation issues: the implementer might not have been careful enough to not leave behind unnecessary code, or the un-initialized variable might be receiving a value later (often in the next line), which might lead to inefficient behavior.

Unused global definitions might mean: (1) there are still functionalities for which there are no tests, (2) some parts of the system are not needed and overcomplicated the system without adding benefits.

Having every type, data and functionality publicly available speeds up the writing of tests, but in the long run this practice can create hard to maintain architectures. Internal representation cannot change after customers started using it, without imposing extra costs on the customers side.

### 3.3.4 Relations to the number of modules

We have measured the size of these projects to see if there is a difference in what ETSI and 3GPP works with. We have found that the number of modules of the 3GPP projects were between 56 and 249, while ETSI projects had 8 to 68 modules. There seems to be a clear separation in size between the projects of the two organizations. 3GPP is working with projects having much more modules and larger network structure.

We also measured the cumulative project risk factors that were defined in section 3.1 (Figure 3.6). According to our measurements the average project risk factor turned out to be 60.075 points. In this case there was no big difference between ETSI and 3GPP developed test suites. The 3 projects with the lowest risk factors are all part of the Intelligent Transport Systems test suites developed by ETSI (relatively new development at that time).

## 3.4 Costs and quality issues

### 3.4.1 Technical debt analysis

After exploring the test system quality issues in the previous chapter our target is to estimate the effort needed to correct them.

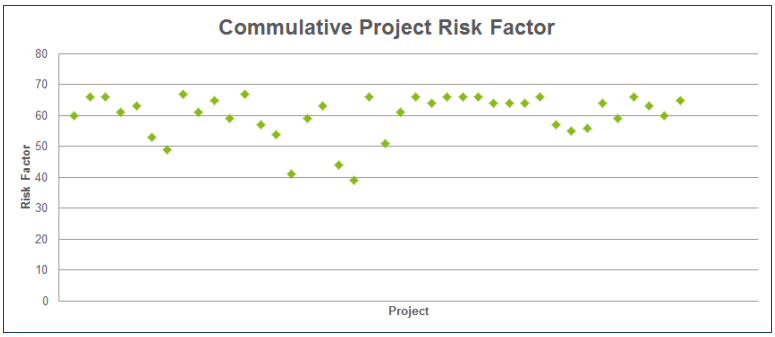



FIGURE 3.6: The cummulative risk factors of the examined projects

### Estimations

First, applying the Delphi method [119], estimates were collected on how long a single instance of a given code smell type correction would take.

We gathered data from 10 experts in the field of test software engineering at our industry partner. The team consisted of a test system architect, test system developers and engineers working in maintenance & support.

In order to address the difficulty level of the issues we did 3 estimates for each code smell type[9]:

- Easy: The issue has only local effects if any, the context tells the original intent and there is no need to change external systems[10].
- Average: A scenario that best fits the experts daily experiences.
- Hard: The issue may affect other files or semantic contexts, the context is not helpful in solving the issue and external system might be affected[11].

We used the following estimation process:

1. Each member of the group gave an estimate.
2. The group was informed about the average and distribution of the estimates.
3. Those giving estimates in the lower quartile and in the upper quartile were asked to tell the rest of the group why their estimates were as they were.
4. The group estimated again. That time taking the previous results and the provided arguments for the "extreme" estimates into account.
5. This might continue two, three, four, or more times until the variation in the estimates was sufficiently small. In our experiences, the variation decreased rapidly. This gave confidence in the final estimation result.

[9] We consciously left out cases, where the work might disrupt other developers work. We also did not address issues created by processes.

[10] For example: in a small function a local variable is not used.

[11] For example circular importation as a structural issue: the structure of the code might need to change, the reason of existence might not be documented, and the change of the code might require changes that have to be documented.

The arithmetic mean of the numbers was calculated and rounded to $0.5$ precision.

#### Estimation results

We summarize the results in Table 3.1.

#### Analysis of the estimations

We have observed that some of the code smell types are very easy to fix. In the best case scenario the rounding to $0.5$ leads to 0 hours of effort needed. Estimations for the average case are close to the easy case. The average case is reaching the arithmetic mean of the easy and hard case estimations only in a few cases, and never exceeds that. In most of the cases the average case costs only $0.5 - 1$ hour more effort to fix than the easy case.

According to the estimations, in the daily experience of our experts, most code smells are rather easy to fix.

### 3.4.2 The cost of fixing standardized test suites

Applying the estimated correction times we were able to calculate the technical debt of both 3GPP and ETSI projects (Table 3.2).

We found that standardized test suites have substantial technical debt. In the average difficulty case[12] the technical debt of the projects can be measured on $1000$ Mhr base meaning several man-years of technical debt.

### 3.4.3 Validation

Some of the projects contained syntactical and semantic errors (chapter 3.3). In order to be able to measure technical debt we had to correct these issues. Depending on how the official corrections of these issues will be done the measured numbers might differ slightly.

Projects, marked with asterisk in Table 3.2, have incomplete archives or import modules of non TTCN-3 or ASN.1 kinds, that are not supported currently by our tool. In those modules the correct number of the founded issues could be higher.

[12] All detected code smell instances assumed to require average amount of work to solve

TABLE 3.1: Estimated cost of fixing code smell types (Mhr)

| Smell | Easy | Average | Hard |
|---|---|---|---|
| goto | 1 | 5.5 | 26 |
| circular importation | 2 | 12 | 80 |
| missing imported module | 0 | 0.5 | 3.5 |
| unused module importation | 0 | 0.5 | 1 |
| non-private private definitions | 0 | 0.5 | 4.5 |
| visibility in name | 0 | 0.5 | 4.5 |
| unnecessary negation | 0 | 0.5 | 3.5 |
| module name in definition | 0 | 1 | 3.5 |
| type in definition name | 0 | 1 | 2 |
| magic constants | 0 | 0.5 | 3 |
| infinite loops | 0 | 1 | 3.5 |
| uninitializaed variable | 0 | 0.5 | 2 |
| size check in loop | 0 | 1 | 5 |
| consecutive assignments | 0 | 1 | 6 |
| read-only variables | 0 | 2 | 5 |
| too many parameters | 1 | 3 | 37 |
| too complex expressions | 1 | 2 | 8 |
| empty statement blocks | 0 | 2 | 5 |
| too many statements | 2 | 6 | 50 |
| too big/small rotations | 1 | 2 | 8 |
| conditional statement without else | 0.5 | 1 | 8 |
| switch on boolean | 0.5 | 1 | 2 |
| setverdict without reason | 0.5 | 1 | 2 |
| uncommented function | 0.5 | 1 | 3.5 |
| stop in functions | 0.5 | 2.5 | 50 |
| unused function return values | 0 | 0.5 | 9.5 |
| receive accepting any value | 0.5 | 1 | 6 |
| insufficient altstep coverage | 1 | 5 | 76 |
| alt that should use alt guards | 1 | 2 | 8 |
| alt that should use templates | 1 | 2 | 8 |
| shorthand alt statements | 0.5 | 5 | 50 |
| isbound condition without else | 0.5 | 1 | 8 |
| Non-enumeration in select | 0.5 | 3 | 8 |
| Insufficient coverage of select | 1 | 5 | 15 |
| Iteration on wrong array | 1 | 5 | 20 |
| unused module level definitions | 0.5 | 4.5 | 18 |
| unused local definitions | 0 | 0.5 | 1.5 |
| unnecessary controls | 0.5 | 1.5 | 5 |
| unnecessary 'valueof' | 0.5 | 1 | 5 |

TABLE 3.2: Estimated technical debt in test suites (Mhr). Projects: 3GPP EUTRA(1), 3GPP IMS(2), WiMAX/HiperMAN(3), WiMAX/HiperMAN 1.3.1 (4), ePassport Readers(5), Session Initiation Protocol(6), IP Multimedia Subsystem(7), IPv6(8), Digital Private Mobile Radio(9), Digital Mobile Radio(10), Intelligent Transport Systems(11). Project identifiers refer to data at `www.ttcn-3.org`

| Project | | | | |
|---|---|---|---|---|
| No. | Identifier | $Min$ | $Avg$ | $Max$ |
| 1 | 36.523-3v10.3.0 | 1528 | 20659.5 | 91282.5 |
| 2 | 34.229-3v9.7.0 / IMS34229 | 392 | 4053.5 | 16886 |
| | 34.229-3v9.7.0 / IMS36523 | 580.5 | 6767 | 30392.5 |
| 3 | TS 102 624-3 | 1699 | 13262 | 63426.5 |
| 4 | TS 102 545-3 | 2552 | 14979.5 | 69307 |
| 5 | TR 103 200 | 163 | 1928.5 | 8949.5 |
| 6 | TS 102 027-3 | 1335 | 7126 | 39363 |
| 7 | TS 101 580-3* | 833.5 | 7438 | 33715 |
| | TS 101 606-3* | 307.5 | 2979.5 | 13382.5 |
| | TS 102 790-3* | 729.5 | 6529 | 28956.5 |
| | TS 102 891-2* | 705.5 | 6237.5 | 28136 |
| | TS 186 001-2 | 844 | 9179 | 40899 |
| | TS 186 001-4* | 557 | 5459 | 24966.5 |
| | TS 186 002-4 | 1326.5 | 12378 | 52104.5 |
| | TS 186 002-5 | 856 | 10703.5 | 42237.5 |
| | TS 186 005-3* | 676.5 | 6058.5 | 27148.5 |
| | TS 186 007-3* | 706 | 6211 | 27998 |
| | TS 186 009-3 | 1005.5 | 9722.5 | 42861.5 |
| | TS 186 010-3* | 706.5 | 6330 | 28587 |
| | TS 186 014-3* | 720 | 7092 | 32606.5 |
| | TS 186 016-3* | 676.5 | 6058.5 | 27148.5 |
| | TS 186 017-3* | 676.5 | 6058.5 | 27148.5 |
| | TS 186 018-3* | 676.5 | 6058.5 | 27148.5 |
| | TS 186 022-3* | 691 | 6093 | 27555 |
| 8 | TS 102 351-3 | 204.5 | 2107 | 9357.5 |
| | TS 102 516 ver 1.1.1 | 352 | 3054 | 13542 |
| | TS 102 516 ver 1.2.1 | 377 | 3347.5 | 14961 |
| | TS 102 516 ver 3.1.1 | 640.5 | 5688.5 | 25697 |
| | TS 102 594 ver 1.1.1 | 497 | 4597.5 | 21407 |
| | TS 102 594 ver 1.2.1 | 527.5 | 5011.5 | 23092 |
| | TS 102 596 ver 1.1.1* | 413.5 | 4334 | 19952.5 |
| | TS 102 596 ver 1.2.0 | 512.5 | 5212 | 24017.5 |
| | TS 102 751 ver 1.1.1 | 517.5 | 5106 | 23234.5 |
| 9 | TS 102 587-4 | 220 | 2512.5 | 10074.5 |
| 10 | TS 102 363-4 | 592 | 4836 | 18359 |
| 11 | TS 102 859-3* | 193 | 2082.5 | 9175 |
| | TS 102 868-3 ver 1.1.1* | 186 | 1652 | 7615.5 |
| | TS 102 869-3 ver 1.2.1* | 187 | 2093.5 | 10218 |
| | TS 102 870-3 ver 1.1.1* | 137 | 1350.5 | 6158 |
| | TS 102 871-3 ver 1.1.1* | 161.5 | 1927.5 | 8796.5 |

# Chapter 4

# Architecture of Test Systems

The aim of this chapter is to analyse the structure of large test systems.

**Thesis 4:** I observed that large scale TTCN-3 test suites show small-world properties and seem to converge to scale-free.

**Thesis 5:** Based on my analysis I was able to show that TTCN-3 test systems contain issues on architectural level and my visualization solution makes it easier to detect these issues comparing to other available solutions.

## 4.1 Structural analysis

The study on structural analysis for test programs is a new concern in the TTCN-3 world. However, there exist approaches for codes in software engineering.

### 4.1.1 Experimental setup

We analyzed the module structure of eleven TTCN-3 based test projects by measuring the incoming and outgoing connections of each module, and creating graphs on the collaborations between them. Some of these projects were standardized, some were industrial.

We measured for each module how many others they import ($I(module)$), and how many times they were imported ($O(module)$) by other modules. Table 4.1 shows the $I_{max}(project)$ (the highest number of modules imported by the same module) and $O_{max}(project)$ (the biggest number of modules importing the same module) values for all projects. Although there is no direct correlation, projects having more modules are more likely to have a higher $I_{max}(project)$ , $I_{max}(project)$ values and more lines of code.

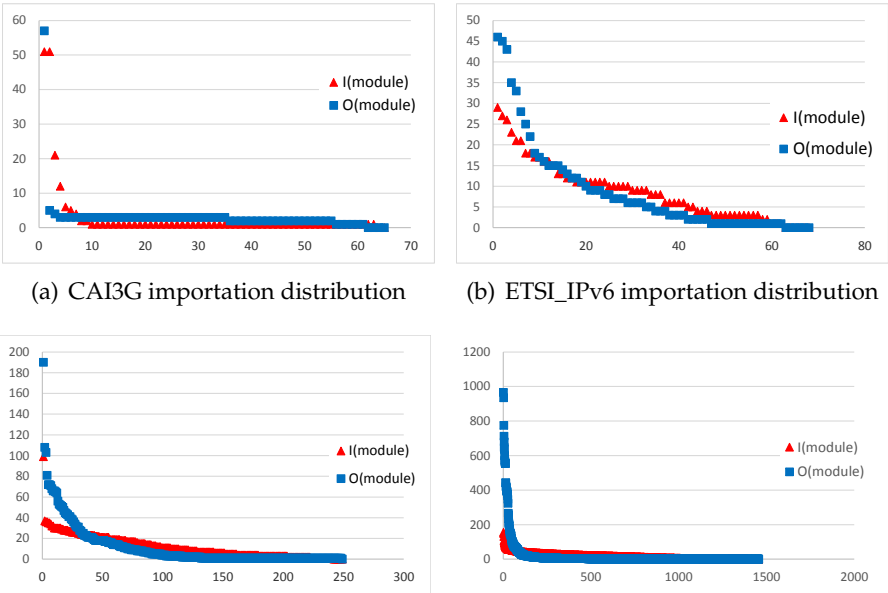
As the size of the projects grows $O_{max}(project)$ becomes larger than $I_{max}(project)$. While $I_{max}$ stays around or below 10%, $O_{max}$ exceeds 40%. In the standardized project *3GPP EUTRA* there is one module imported by 76% of the modules, in the *MTAS* industrial project there is one imported by 66% of the modules.

#### Importation data analysis

Figure 4.1 shows the distribution of $I(module)$ and $O(module)$ values for all of the modules in four projects. In all cases the measured values are displayed in descending order, with the X axis only showing the position

TABLE 4.1: importation data

| Project vs test | Number of modules | $I_{max}(project)$ | $O_{max}(project)$ | LOC |
|---|---|---|---|---|
| TGC_traffic | 20 | 10 | 6 | 127.470 |
| ADC_OMA | 42 | 23 | 8 | 21.174 |
| Hiperman 1.3.1 | 49 | 20 | 41 | 142.867 |
| CAI3G | 65 | 51 | 57 | 53.583 |
| ETSI IPv6 | 68 | 29 | 46 | 67.505 |
| T. A. Wireline | 71 | 15 | 34 | 97.672 |
| W_MOCN | 205 | 36 | 85 | 442.784 |
| 3GPP EUTRA | 249 | 99 | 190 | 246.316 |
| SAPC | 364 | 21 | 149 | 58.199 |
| TitanSim | 920 | 70 | 405 | 1.037.184 |
| MTAS | 1456 | 155 | 966 | 3.000.248 |




(a) CAI3G importation distribution (b) ETSI_IPv6 importation distribution

(c) 3GPP EUTRA importation distribution (d) MTAS importation distribution

FIGURE 4.1: Distributions of importation

of each module in this ordering. There are only a few modules that import many others, or are imported many times, most of the modules import only a few others, often less then five others. The distributions of $O(module)$ and $I(module)$ become smoother as the number of modules increases in the projects.

Table 4.2 shows how well the logarithmic and power trend lines fit the measured data for each project. According to our measurements on bigger projects, in descending ordering, $I(module)$ follows a logarithmic trend line very closely, with $r^2$ values above $0.9$ up to $0.99$; $O(module)$ values in descending ordering, follow a power law trend line, with $r^2$ values above $0.8$ up to $0.97$.

TABLE 4.2: trend fitting

| | log $r^2$ | | power $r^2$ | |
|---|---|---|---|---|
| Project vs test | $I$ | $O$ | $I$ | $O$ |
| TGC_traffic | 0.9 | 0.93 | 0.84 | 0.84 |
| ADC_OMA | 0.84 | 0.95 | 0.85 | 0.82 |
| Hiperman 1.3.1 | 0.65 | 0.88 | 0.47 | 0.77 |
| CAI3G | 0.50 | 0.29 | 0.69 | 0.58 |
| ETSI IPv6 | 0.97 | 0.96 | 0.72 | 0.83 |
| T. A. Wireline | 0.94 | 0.94 | 0.70 | 0.87 |
| W_MOCN | 0.98 | 0.68 | 0.79 | 0.90 |
| 3GPP EUTRA | 0.90 | 0.86 | 0.71 | 0.88 |
| SAPC | 0.95 | 0.47 | 0.72 | 0.96 |
| TitanSim | 0.99 | 0.60 | 0.79 | 0.96 |
| MTAS | 0.97 | 0.49 | 0.65 | 0.97 |

**Project diameter analysis**

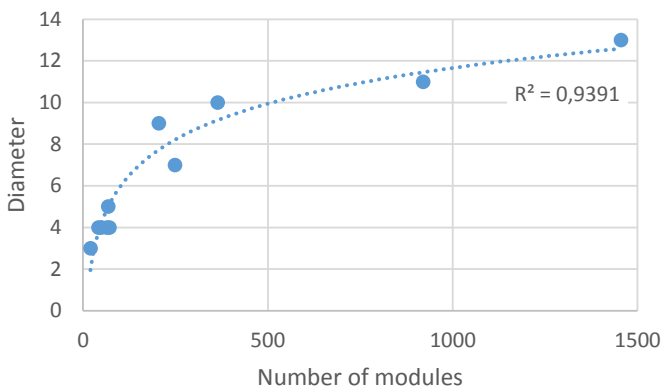



FIGURE 4.2: Diameter of the graphs

In case of TTCN-3 the diameter of the module importation graph (the longest path from the set of the shortest paths between any two nodes in the graph) seems to be a logarithmic function of the number of modules present in the project (Figure 4.2). This is in line with previous observations [120] on small-world and scale-free networks.

We note that this observation does not say anything about the growth of an individual project.

## 4.2 Architecture visualization

In this section we focus on the following questions: (1) which layout and clustering of nodes are the most useful in daily work, (2) using the appropriate layout is it possible to find architectural issues in some available test suites, (3) does visualization tool embedding into the development environment support the daily usage?

### 4.2.1 Experimental setup

We used two graphical interfaces to display the architectures rendered by the JUNG [121] framework. In one window, the satellite view shows a scaled down, black and white version of the whole system. The other (main) window shows a part of the architecture, equipped with graphical features. Both the main and the satellite window can be moved around and resized to fit the user's needs. A click anywhere in the satellite view centers the main view on the clicked part of the architecture.

The main view can be zoomed in and out with the mouse scroll. By holding down the right mouse button moving the mouse moves the viewed region. With the left mouse button it is possible to select one or more nodes, and drag the selected nodes to a different part of the visualized region. Right clicking on a selected node brings up a menu, where the user can select to see metrics measured on the module, select the node and all edges going in and out of it (graying the rest of the graph), or jump to the source code of a module.

The main window has a menu for actions with global effect: (1) changing layout, (2) clustering, (3) exporting the graph, (4) showing circular references and parallel paths, (5) searching for nodes. The highlighted edges are colored with red, while other edges are grayed out.

We implemented two algorithms which are similar to [122].

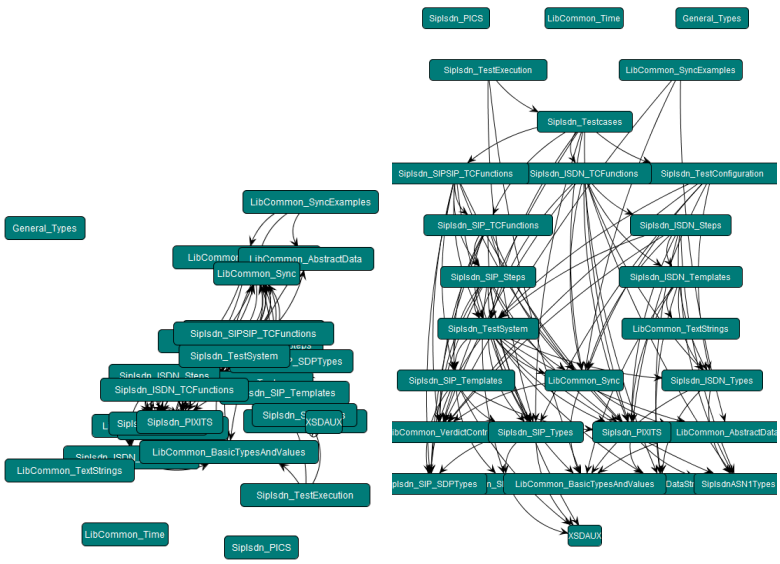



FIGURE 4.3: *IMS Interworking* modules, left: *Fruchterman-Reingold* and right: *DAG* layout

In both cases independent nodes (not imported and not importing) are allocated to the 0-th level. Nodes in strongly connected components are treated as a single virtual node, of which all nodes get on the same level.

Our *DAG* layout algorithm selects the nodes with no incoming edges for the first level. Each further level contains nodes only imported by nodes on the previous levels (Figure 4.3).

Our *Reverse DAG* layout algorithm selects the nodes with no outgoing edges on the first level. Each further level contains nodes importing only nodes from the previous levels (Figure 4.4).

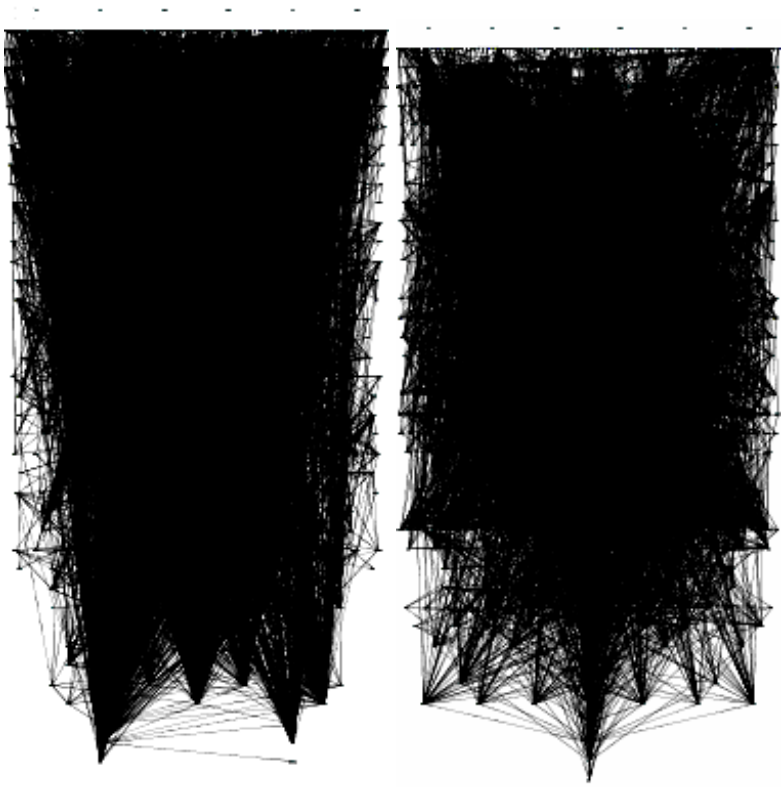


FIGURE 4.4: Industrial test system, left: *DAG* and right: *Reverse DAG* layouts, satellite view

We implemented some clustering algorithms as well in order to be able to reveal more architectural features.

Clustering forms:

1. Grouping: moves nodes that belong to the same cluster close to each other. In this form of clustering it is possible to see the contents of each cluster, to decide if a module should belong there or not.

2. Graph generating: represents each cluster of nodes with a single new node. In this form all previously mentioned layout algorithms are available allowing inspection from several different viewpoints.

Clustering algorithms:

1. Automatically: This algorithm ([123]) automatically creates clusters in the network, detecting the number of clusters to be used for the best representation. In practice this may take very long time (sometimes 10+ minutes).

2. By file location: Files belonging to the same folder are assumed to be in the same cluster. These clusters are represented by the path of the folder on the user interface (users could configure path prefixes to be eliminated from the displayed name).

3. By module name: In this clustering mode the names of the TTCN-3 modules, contained in the source files, are treated as paths. We observed that module names follow a naming pattern: they are made

up of words separated by '_' characters or a switch between small and capital letters[1]. Each module located on the same path (until the last part, assumed to be its name) is assumed to belong to the same cluster.

4. Using regular expressions: In this clustering method the user can decide which modules belong to a cluster, by declaring regular expressions. The modules, whose name fits a given expression, belong in the same cluster.

Please note, that both module location and module name based clustering assumes that there is higher level organizational principle used by the developers. But the TTCN-3 standard does not consider such principles yet.

### 4.2.2 Case study: test suites from ETSI

We have analyzed all test suites (40) publicly available at ETSI's official TTCN-3 homepage `www.ttcn3.org`. Most of the test suites were created by ETSI, some by 3GPP.

ETSI test suites have 8-68 source files and our *DAG* layout found 5 to 15 layers. 3GPP test suites have 56-249 source files and 15 to 18 layers. In these test suites we found several architectural problems.

1. Potentially unnecessary modules
   - We found several files independent from the rest of the test suite[2] (Fig. 4.3).
   - Many test suites had top level files, which might not be needed[3] (Fig. 4.3).
2. Cycles
   - We found one test suite with import cycles among files: `IP Multimedia Subsystem TS 101 606-3`.
   - Several test suites had import cycles among their folders [4] (Fig. 4.5) and among the packages derived from their module names[5] (Fig. 4.6).

[1]For example: IMS_CommonProcedure_Registration, CDMA2000_Templates, EUTRA_CommonDefs

[2]WiMAX test suites; Digital Private Mobile Radio; all Intelligent Transport Systems test suites; all IP Multimedia Subsystem/IMS Supplementary Services test suites + IP Multimedia Subsystem/Network Integration Testing (TS 186 001-2, TS 186 001-4) + IP Multimedia Subsystem/SIP-ISUP Interworking (TS 186 009-3)

[3]The name of the files does not have 'testcase' or 'main' in it. For example: LibCommon_Time, ePassport Readers TR 103 200, IP Multimedia Subsystem (TS 186 001-2, TS 186 002-4, TS 101 580-3, TS 102 790-3, TS 186 007-3, TS 186 014-3, TS 186 022-30), all IPv6 tests suites

[4]all IPv6 test suites, Intelligent Transport Systems (TS 102 859-3, TS 102 868-3, TS 102 870-3, TS 102 871-3), IP Multimedia Subsystem (TS 186 001-2, TS 186 002-4, TS 186 009-3), all 3GPP test suites

[5]all IPv6 test suites, IP Multimedia Subsystem (TS 186 014-3, TS 186 010-3, TS 102 891-2, TS 102 790-3, TS 186 016-3, TS 101 580-3), Intelligent Transport Systems (TS 102 859-3, TS 102 868-3, TS 102 870-3, TS 102 871-3), all SIP-ISUP Interworking test systems, ePassport Readers TS 102 624-3, WiMAX TS 102 624-3, all 3GPP test suites

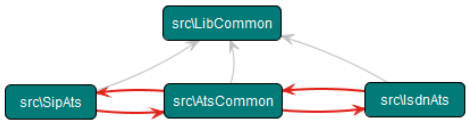



FIGURE 4.5: *IMS inteworking (TS 186 001-2)*. Clustered by folders, the circles are shown.

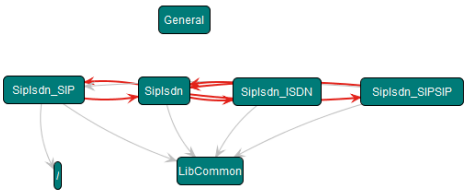



FIGURE 4.6: *IMS inteworking (TS 186 001-2)*. Clustered by packages, the circles are shown.

### 4.2.3 Case study: an industrial test suite

An industrial test suite, mentioned here, contains 882 files, displayed in 39 layers[6] by our *DAG* layout.

There is a clear difference in size and complexity between test suites found in standards (Fig. 4.3) and in the industry (Fig. 4.4).

We organized a one day event where future users could try our tool on their systems. The aim of this day was to improve the internal quality of their system by reviewing and reducing the number of problems reported. Two architects revealed 57% of the reported circular dependencies resulting in a 3% improvement in the build time of the whole system.

### 4.2.4 Validation

We run a survey (see Appendix C.2) with three test system architects at our industry partner, who gained experience in using our tool.

- All respondents reported that our *DAG* layout was the easiest to understand and most useful in practice.
- One architect reported not to have separated modules in his project. Two reported *DAG* and they found the potentially unnecessary modules very easy.
- Everyone found the first level[7] very useful: unused parts of libraries became visible.
- One architect could not evaluate the visualization of circles: there were none in his project. Two architects reported that the visualization of circles was very useful.
- Two preferred the graph generator for practical work, one had not used them.

[6]The diameter of this network is 11.
[7]Populated with modules which are not imported.

- The direct location based clustering was found useful in one case, in revealing the structure of a library system.
- Everyone reported the module name based clustering to be useful. It could be used for checking correct naming convention usage.
- For the question "How important is it for you, that these tools are integrated into the development environment?", we received the answers "I would not use it otherwise" (2 times), and "makes it easier to install and use. Immediate feedback after change is more useful in smaller projects".
- One of the architects reported that he needed only 3-4 tries to figure out how to operate the main and satellite views and proposed to have some pop-up window that lists the controls for ten seconds. Others found the views immediately usable.
- Everyone reported that the dependency visualization is the most useful during reviews.

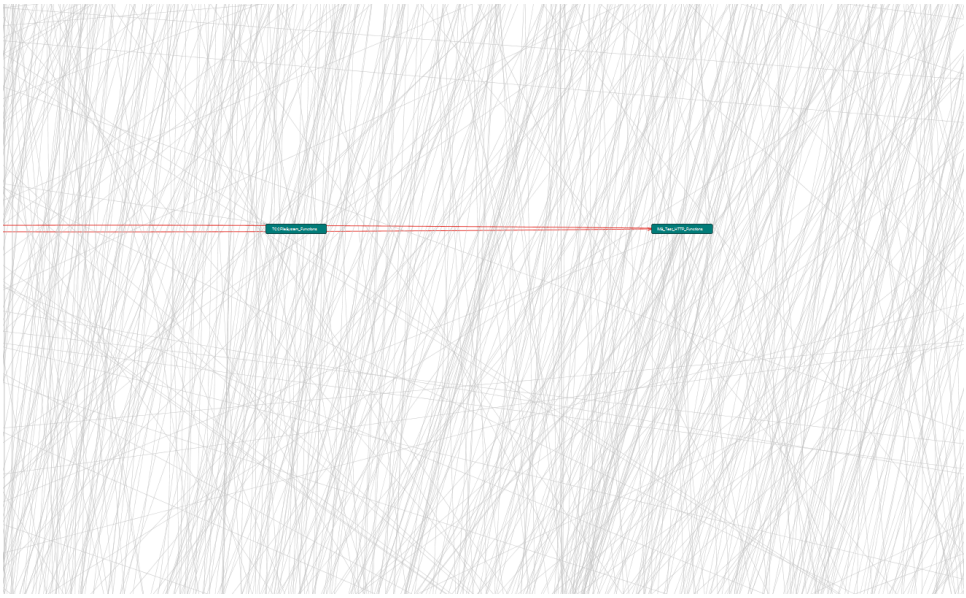


FIGURE 4.7: Industrial test system, *DAG* layout, detected circles.

In the following we show the answers to our questions stated in the beginning of this section in a concise form.

**Question 1:** Is our layered layout better than the existing layouts for daily work?

Respondents to our survey (see 4.2.4) indicated that for their daily work they find the layered layouts (Fig. 4.3, 4.4) better, than *Fruchterman-Reingold* [7] and *Kamada-Kawai* [6] layouts.

**Question 2:** Are clustered layouts useful in daily work?

The module name based clustering was reported to be useful for checking naming conventions. The location based clustering could be used to reveal library structure.

**Question 3:** Do available test suites contain architectural issues?

Sections 4.2.2 and 4.2.3 show that several TTCN-3 test suites contain architectural issues: import circles, files independent from the rest of the test suites, potentially unnecessary top level files.

**Question 4:** Are tools embedded in the development environment preferred to external tools?

Our respondents preferred integrated tools mentioning that they are easier to install and provide immediate feedback.

# Chapter 5

# Quality evolution of test systems

In this chapter we show empirical observations on the evolution of two large industrial test systems. We monitored the development of these systems and measured their code quality characteristics for a five years period.

**Thesis 6:** I observed that the internal quality evolution of the examined TTCN-3 test systems follows a predictable pattern similar to that of programming languages and projects.

## 5.1 History of the studied systems

In this section we show the background and historical information on the observed systems.

Current test systems may have many different parts, which might be developed separately in different organizations. Although these parts are designed to become test suites or serve as components of test suites, most of them can not be called tests (e.g. software layer converting between abstract TTCN-3 messages and actual bit stream messages). For this reason in this chapter we use the term "test system" to describe software components of test suites and the test suites built of them.

We have studied two test systems developed and used at our industry partner. The history of these systems goes back to 2005. We started to analyze them in 2012. At the end of 2012 the two systems were merged to form a single solution.

Both test systems are built on a set of libraries and tools in a hierarchical structure. We will call this set of systems `Common`. Parts of `Common` in the lower abstraction layers support (1) sending and receiving messages of a specific protocol, (2) the protocol logic (3) and the forming of a glue layer between a generic product and some specific usage.

`System-1` was originally designed for demonstrating and testing the features of `Common`, containing a set of project independent, reusable data structures and algorithms that can be used for creating high levels of load in TTCN-3.

`System-2` was aimed at testing IMS[1] products. At the end of 2012 these two test systems were merged into one, which we will call the `Merged` System.

[1] IP Multimedia Core Network Subsystem is an architectural framework designed by 3GPP for evolving mobile networks beyond GSM

`System-1`, `System-2` and `Merged` offer complex and computationally intensive functionalities. They are used to test if the System Under Test is able to: (1) handle large amount of users, (2) handle large data traffic coming in a mix of several supported traffic type and (3) stay stable for long durations (days or even weeks).

In the following we provide a list of the most important events which could have influenced the quality of the studied systems.

- 2005 - 2006: The development on Core Library started.
- Mid. 2007: First Core Library release.
- Early 2008: `System-1` was born. Developers were dedicated to independent customers with little coordination among them.
- Mid. 2009: A team in `System-1` switched to Scrum methodology led by an experienced scrum master. Strong coordination was manifested for the teams but there were still external developers working on the same source codes.
- End of 2009: The scrum master moved to a different unit inside the company. Her place was filled with people she trained earlier.
- 2010: `System-2` was moved from abroad to in-house. The in-house team decided to rewrite the code from ground up.
- 2010 - 2011: The team of `System-1` was experimenting with Kanban and custom methodologies designed specifically for the project.
- February 2012: Work starts on Titanium.
- 2012 beginning: `System-2` changed to a new version handling repository. This was the first version of its source code available for us to study.
- 2012 first half year: New scrum master and product owner were selected for `System-1`. One system architect was selected from each team to analyze requirements, write implementation studies and guidelines. A System Architect Forum was created, fostering information sharing between system architects.
- 2012 second half year: The organizational structure of `System-1` was changed. The scrum master and the product owner were replaced. From this point in time there were no external developers changing the source code in parallel with the team.
- Dec. 2012: `System-1` and `System-2` were merged forming the `Merged` system. The source codes were stored in a new source code repository.
- May 2013: during a "Boost day" event Titanium is integrated into the continuous integration server of `Merged`. The effect of every change is measured and displayed on web pages accessible by all developers and managers in the project.

- 11 July 2013: "Titanium Quest" was organized. Among others, the participants removed 10% of FIXME and TODO comments, reduced the number of "circular importations" by 57% and the number of "unused imports" by 50%. The removal of the circular imports enabled a 3% improvement in the build time of the `Merged` System.
- 2014 first half year: All of the system architects of the `Merged` system are replaced by a single system architect.
- 17 July 2014: The "Green Day" event is organized. Among others, most of the remaining "unused imports" were removed.
- 4th December 2014: the "Black Thursday" event is organized. Participants removed 0.6% of the code, reviewing read-only variables, inout and out parameters, unused local definitions

"Titanium Quest", "Green Day" and "Black Thursday" were 24 hour code fixing challenges.

From organizational point of view these systems were developed by several teams. The size, structure and responsibilities of the teams changed with time. All teams were working within the same organizational unit, sitting together in the same part of the building. Communication among members of teams and among teams was not obscured.

Developers of `System-1`, `System-2` and `Merged` mentioned that between 2008 and 2011 the system architect was always available for questions but it was not mandatory to ask him. Members of the System Architect Forum mentioned that they had no tools to enforce their proposals as the teams were following agile methodologies (particularly Scrum) where reviewing and accepting the implementations of features/requirements was the responsibility of the PO role.

Between 22 July 2013 and 17th July 2014 there were 73 issues reported for the `Merged` System. These issues range from product and structural issues via performance and code duplications to code complexity and inefficient variable scoping. All reports contained the location and a description of the specific defect. Some reports contain advises for possible corrections as well.

During 2014 we organized trainings to spread knowledge about code smells with the following agendas:

- January: Handling lists efficiently in TTCN-3,
- Mids of February: Introduction to code smells and their relevance,
- End of February: Advanced uses of Altsteps
- March: How to efficiently assign a value?
- April: Parameter passing in TTCN-3 in theory and practice.

Table 5.1 shows the actual efforts (in ratios of man-hours) reported for the test systems at different points in time. For each year we show data for the months January and October[2] to represent the starting and closing of the year.

[2]In November and December employees tend to go on vacations, significantly changing the amount of work reported on each project.

| Name | 2009 | | 2010 | | 2011 | | 2012 | | 2013 | | 2014 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Jan | Oct | Jan | Oct | Jan | Oct | Jan | Oct | Jan | Oct | Jan | Oct |
| `Common` | 1.00 | 2.06 | 1.70 | 1.92 | 1.54 | 1.97 | 1.90 | 1.56 | 1.30 | 1.50 | 1.39 | 1.36 |
| `System-1` | 1.20 | 0.52 | 0.64 | 0.76 | 0.76 | 0.78 | 0.81 | 1.14 | | | | |
| `System-2` | | | | 0.68 | 0.42 | 1.07 | 1.06 | 1.13 | | | | |
| `Merged` | | | | | | | | | 2.63 | 2.65 | 3.35 | 3.51 |

TABLE 5.1: The actual effort (ratios of man-hours) reported on the investigated systems at different points in time. The values are shown as ratios compared to the effort reported for `Common` in January, 2009.

The efforts invested into the products show a growing trend with some fluctuations. Since the work started in 2009 the number of man-hours reported for the project have doubled by the end of 2014.

After the merge all previous efforts invested into `System-1` and `System-2` were redirected to `Merged` taking away some resources from `Common`.

## 5.2 Code smell measurements

In this section we present our measurements. For each day in the investigated range we checked out the source code in the state it was at midnight and measured the number of code smells (listed in Table B.1) present.

### 5.2.1 Size

We analyzed the size growth of the `System-1` and `Merged` systems measured in LOC. Figure 5.1 shows the measured data, a quadratic trend line fitted, and the Lehman's prediction according to equation (2.5.1). The maximal absolute error between the measured data and the predicted model is about 3%.

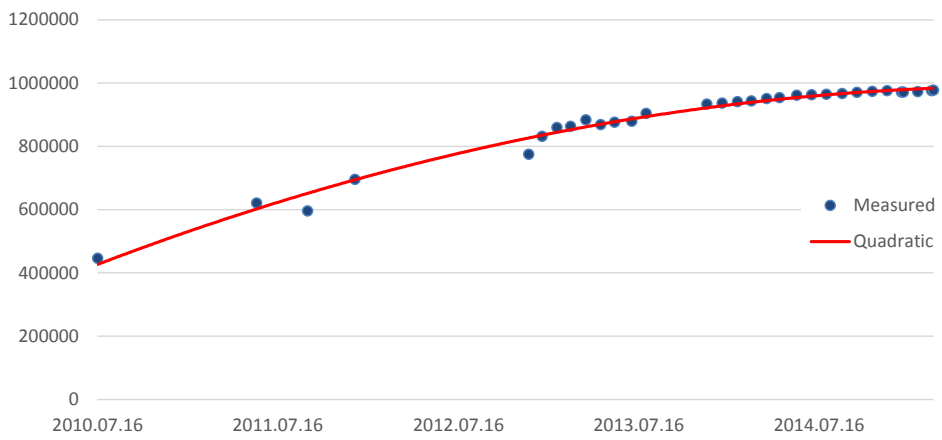



FIGURE 5.1: Size evolution of the `System-1` and `Merged` systems.

### 5.2.2 Correlations among code smells

For each possible pair of code smells we calculated the Pearson correlation between the data series of the code smells on the `Common + System-1 + Merged` system evolution (Table B.1). We excluded code smells having less than 50 occurrences at the measuring points which may break the trends. Based on the correlation values the code smells could be separated into 3 correlation groups:

1. In the largest group, the correlation was at least 0.95 between the smell pairs. These are exactly the code smells that have never been addressed during special events: *FIXME tags, TODO tags, empty statement block, if instead altguard, magic numbers, magic strings, logic inversion, definition should be private, read-only inout formal parameter, size check in loop, switch on boolean, too complex expression, too many parameters, uncommented function, uninitialized variable, unused function return values, visibility in definition*.

2. Code smells with correlation values related to the first group, lying between 0.3 and 0.95, were addressed during special events, but only a fraction of their appearances were removed: *module name in definition, if without else, unnecessary control, read-only local variable, typename in definition, unused global definition, circular importation*.

3. Three code smells have zero or negative medium correlation values ($-0.42$, $-0.72$ and 0.04) compared to the members of the first group. Most of the occurrences of these code smells were addressed during special events or in personal efforts: *readonly out formal parameter, unused import, unused local definition*.

### 5.2.3 Code smell trends

In this section we show how the different events in the history of the test systems have correlated with the changes in the number of code smells.

#### First correlation group

From the first correlation group we present the *magic strings* code smell. The data series of other code smells from this group have high correlation with this data series, hence, we omit to show them.

In both systems the cumulative number of magic strings was increasing following a nearly linear trend (Figure 5.2). Before the merge the number of *magic strings* was growing by 5152/7923/7027 instances in `System-1` and by 4225 instances in `System-2` per year. Direct after the merge the growth dropped to 2378 instances per year for most of the year 2013. The growth speed reached 4733 instances per year in 2014.

It is interesting to point out that the reduction of growth after the merge, lasted approximately until the numbers were fitting to the original growth trend of `System-1`. From 2014 the growth of `Merged` followed a trend much closer to that of `System-2` than to `System-1`.

The sudden increases in the measured data in `System-1` till the middle of 2011 indicates 3 months development cycles and developers working on branches separate from the main development branch. Later in `System-1`

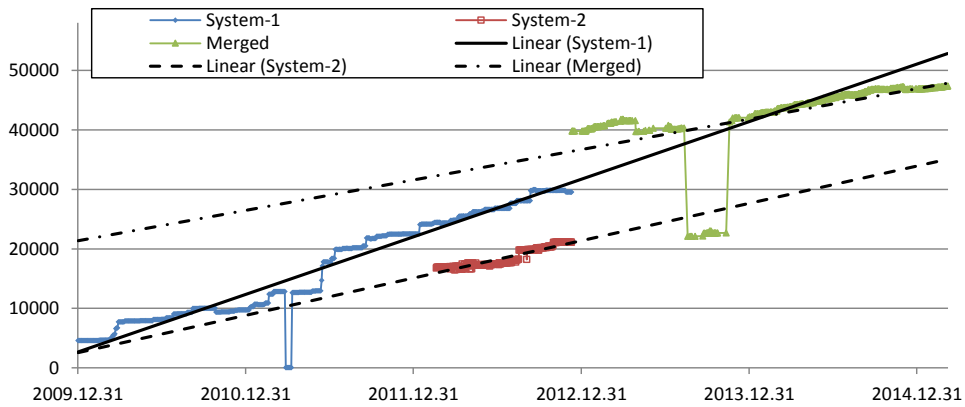



FIGURE 5.2: *Number of magic string* issues and its linear approximations.

and `System-2` these increases are not present, indicating frequent changes to the main development branch. This fits to the part of the history: the development was not done as a team, but rather individuals serving the needs of separate customers.

Between April and May 2011 the number of most code smells in this group temporarily dropped. The project descriptor was corrupted in both cases. The build system used a forgiving way for extracting information from the project descriptor, but for our tool this made the project appear as if large amounts of files were removed. At the end of 2013, already after agile and continuous integration was introduced, the same problem reappeared while code quality measurements were displayed in publicly available places.

### Second correlation group

From the second correlation group we show each code smell separately.

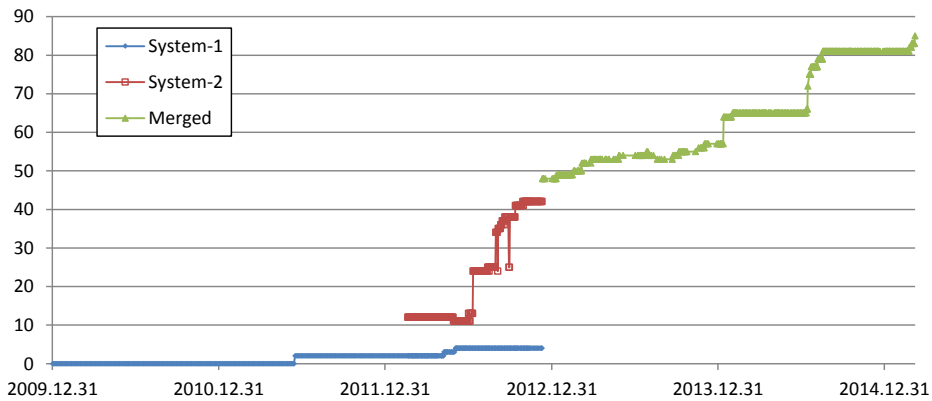



FIGURE 5.3: *Module name in definition* smell trends

In case of the *module name in definition* code smell (Figure 5.3) the trends of `System-1` and `System-2` seems to be added together, and following the growth trend of `System-2`.

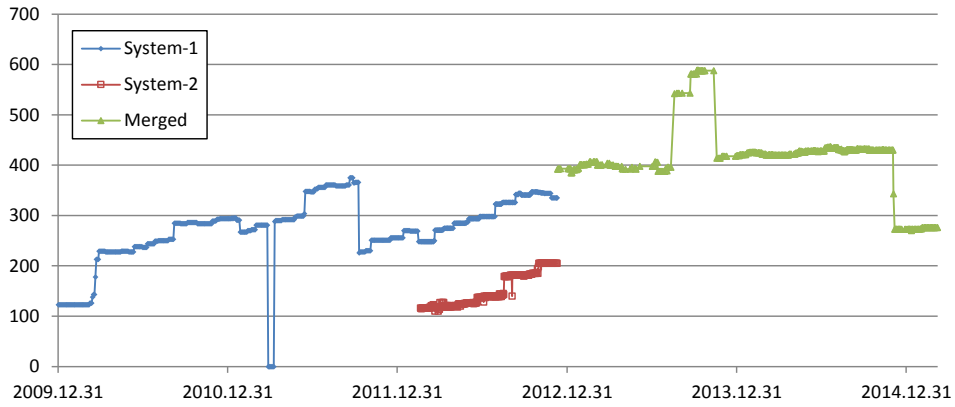



FIGURE 5.4: *Readonly local variable* smell trends

In case of the *read-only local variable* code smell (Figure 5.4) the growth trend slowed down after the merge, creating a different trend from that of its source systems. In `System-1` the growth was 118 instances in 2012, and 89 in `System-2`. The trend continued by 9 in 2013 and 11 in 2014 after the merge until the occurrences were greatly decreased at the "Black Thursday" event.

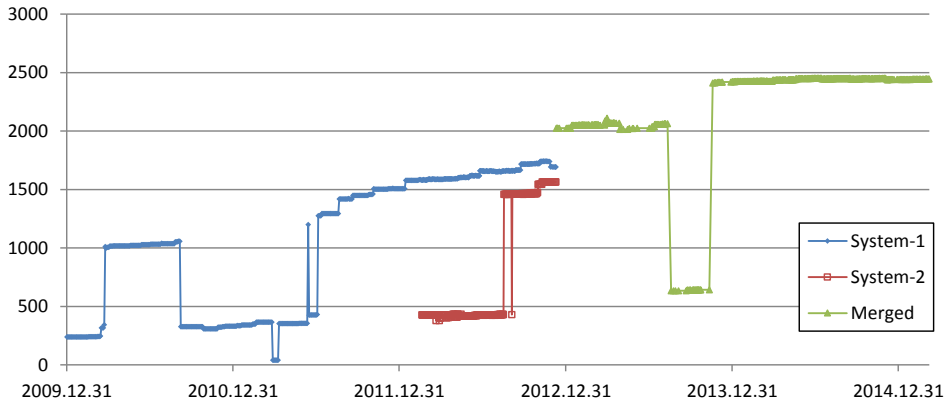



FIGURE 5.5: *Typename in definition* smell trends

The *typename in definition* trends (Figure 5.5) also slowed down after the merge. The reason behind the drop in `System-1` from around mid 2010 till mid 2011 was a naming convention change.

In the case of the *unused global definition* code smell the trends in `System-1` continued in `Merged` (Figure 5.6) also slowed down after the merge. Several instances of this code smell were handled during the "Green Day" and "Black Thursday" events. The corruption of the project descriptor caused a temporal drop in April 2011, and a temporal increase at the end of 2013. In the first case files containing *unused global definitions* disappeared from our measurements, in the second case the files disappearing caused the increase in the number of *unused global definitions*.

*Circular importation* followed a different behavior. In `System-1` the occurrences were rare and stable. In `System-2` their occurrences were higher and changing frequently (this smell was reported for every module in the

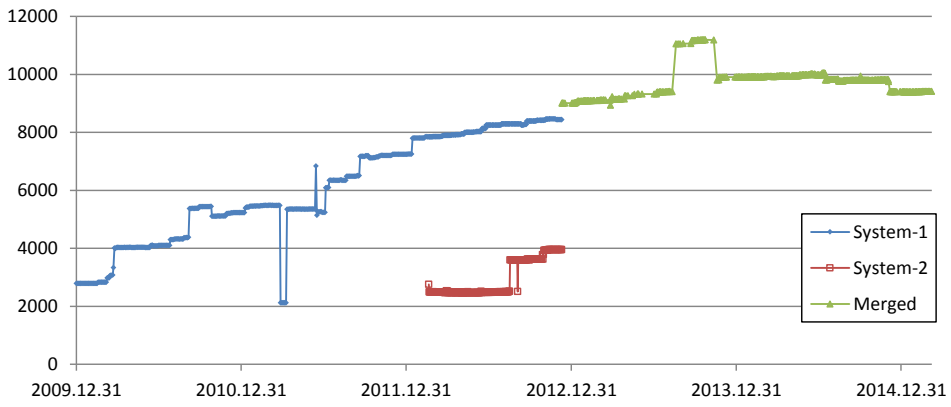



FIGURE 5.6: *Unused global definition* smell trends

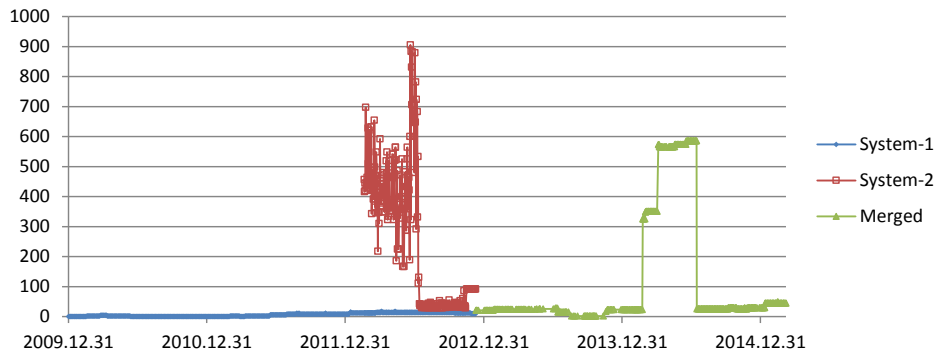



FIGURE 5.7: *Circular importation* smell trends

circle individually in our tool, allowing for small changes in the source leading to large changes in reported numbers of this smell). After the merge the trend stabilized.

In `System-1` the growth was 4 instances in 2012, the growth behaviour was "chaotic" in `System-2` till the half of that year. The growth continued with 2 instances in 2013 and with 7 in 2014 after the merge. When two libraries developed on separate branches were merged in February and March 2014, the numbers increased to 351 and 566. The number of occurrences was reduced to 45 during the "Green Day" event.

The code smells *read-only local variable*, *circular importation* and *unused global definition* were addressed on special events, but only a portion of their numbers could have been corrected.

### Third correlation group

From this group we show only the *unused imports* smell trends.

The occurrences of this smell in `System-1` drops from 1717 to 1398 between June and July and to 215 till the end of December 2012 (Figure 5.8). In `System-2` the occurrences of *unused imports* falls from 420 to 298 on October and to 215 on December, 2012. We found that all of these code quality

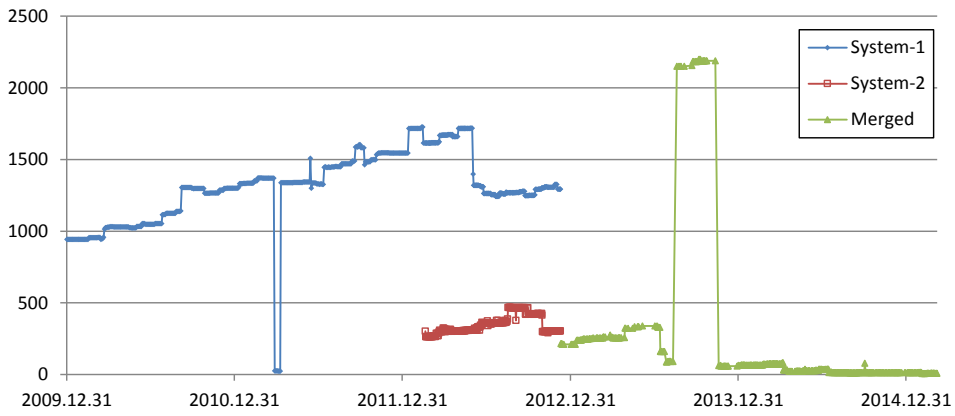



FIGURE 5.8: Number of *unused imports* smell trends.

improvements were related to one employee. After learning that Titanium had support for detecting unused imports she/he decided to clean up some of the code.

Shortly after July 2013 the occurrences of *unused imports* drops from 329 to 84 during the "TitaniumQuest" event.

The large fallback at end of 2013 appeared as an increment of issue numbers: the imports to missing modules were reported as unused.

## 5.3 Trend analysis

In this section we analyse the factors which might influence the quality trends.

- **The number of measured code smells was not affected by the introduction of continuous integration.**

  Continuous integration was introduced together with Agile. The fine tuning of CI took months. Quality gate was introduced into continuous integration during the "Boost day" (May 2013), with the integration of Titanium. We found no direct connection between the number of code smells present in the source code and the introduction of quality control to continuous integration, or continuous integration itself. Most of the observed code smell occurrences followed the same or similar trends after continuous integration was introduced.

  We also observed two cases when project descriptors were corrupted (one before, one after continuous integration was introduced). In neither of the cases did the build and test system notice the corruption. Although during the second case, the code quality displays, driven by continuous integration, showed the changes, they did not evoke immediate action.

  Our experience on the influence of using continuous integration aligns with earlier published results of others ([94, 96, 106]).

- **The number of measured code smells was not affected by the introduction of tool support itself.**

We have created Titanium to detect and report internal quality issues. Titanium was integrated into the continuous integration system during the "Boost day" (May 2013). We have organized tutorials: we explained (1) the usage of the tool, (2) the meaning of the reported code smells and (3) what kind of problems the smells can create. In order to reduce the entry barrier of correction we analysed the observed systems and reported some issues found together with a guide on what to correct, where and how. 73 issues were reported between July 2013 and July 2014 (one year interval) as improvement proposals.

We have found no evidence, breaks in the trends, showing that tool support in itself motivates project members to clean up their code. Yet, measurements show that, when personal motivation is present, or special events are organized, tool support increases productivity. One person can review and correct numerous of instances of issues otherwise unnoticed. These results align with the earlier results of others ([96]).

- **The number of measured code smells was affected by the merging of two test systems.**

  We measured that the merge increased the amount of code smells present and also decreased their previous growth rate. These results align with the fifth law of software evolution ([83]) and other earlier results ([94, 96, 106]).

  It is interesting to note, that the growth of the merged system is between the original growths of the two systems it consists of.

- **The number of measured code smells was not affected by the different development methodologies.**

  During the history of the observed projects the development was performed sometimes by individuals, sometimes by teams. Teams used company specific methods in the beginning, Scrum and Kanban for some time, tailored Agile-like methods for other periods of time.

  We have seen that before the middle of 2011 the changes in the numbers of code smells indicated 3 month development period. After this time the changes became smaller and more frequent. Although this might indicate an effect custom methodologies or maturing in Agile methodologies might have had, there was no change in the general trend lines. The changes became more frequent, but followed the same trends in their effects. Other than the changes becoming more frequent we were not able to find any change correlating to the methodologies, or lack of in our measurements.

- **The number of measured code smells was not affected by changing leaders of the projects.**

  Conway's law [124] suggests that there is a mirroring effect between the structure of an organization and the structure of the product it creates. In our case there were several organizational changes on the lower levels: teams were formed, team internal processes were changed, system architects were appointed, product ownership changed.

In the measured data we were not able to find any evidence that could be related to these changes. We assume that changes in the immediate leadership were not able to affect the systems. The reason for this is not clear: there could be higher-level organizational structures that binded the immediate leaders, or code smells and lines of code might not correlate with such structures.

Based on the information we collected from the system architects and developers we believe the former assumption. There were no organizational tools in place for enforcing the system architect's guides. Tasks were selected for implementation and prioritized for dedicated developers by the distinct customers they support. This relation might have circumvented the power of technical and managerial leaders.

- **Code smells in the observed test system followed predictable patterns during the system's evolution.**

  In the following we show how our findings detailed before relate to Lehman's laws of software evolution ([83]).

  - Our measurements support the 2nd law: in all examined test systems all code smells measured followed an increasing trend unless work was done to reduce them.
  - Our measurements support the 4th law: the work rate in each test system studied stayed approximately the same during their whole lifetime. The invariant work rate was not significantly affected by the changes in history. Lehman showed [125] that although corporate and local management certainly has control over resource allocation and activity targets their ability to do this was constrained by external forces, like the availability of personnel with appropriate skills and trade unions.
  - Our measurements support the 5th law: the average incremental growth of successive releases was largely invariant. This property was not affected by most of the changes in history. Only individual efforts and the merge of the two systems has disturbed the trends. Lehman conjectured [89] that this effect is caused by the rate of acquisition of the necessary information by the participants.
  - The 8th law is usually proved with showing ripples in the measured data, which are believed to reflect self-stabilization through positive and negative feedback. We believe that the slowdown right after the merge was the result of this feedback mechanism. The merge of the test systems increased the amount of code to be maintained and developed further, but at the same time, the growth trends were somewhat decreased.

# Chapter 6

# Human side of quality

This chapter contains the result of a survey. We surveyed individuals working in software development projects. We wished to understand how the knowledge of IT employees differs having various roles (manager, developer, tester, technical writer), how they gain new knowledge, how they vary in thinking about their processes and anti-patterns in software development. This chapter presents the results of the survey focusing on roles, experience levels and the size of the companies respondents were working for. Our main research questions were:

- How well known are the techniques of different fields?
- How important are the different mindsets?
- What are the main sources of new knowledge?
- How useful are the several knowledge gaining methods in daily work?
- How different is the way of thinking in the various roles?
- How are anti-patterns perceived?
- Are people motivated and supported to resolve anti-patterns?
- How does the size of the company or team organization impact people's knowledge, thinking and perception of anti-patterns?
- How does experience level impact people's knowledge, thinking and perception of anti-patterns?

**Thesis 7:** I observed that the mindset of testers and developers is similar. To be more precise I showed that from human aspects regarding the internal quality a test project is very similar to a software project.

## 6.1 The survey

In our survey we investigated the knowledge and concerns of people working in software development projects.

Our main goal was to explore the thinking of software professionals working in different roles, to gain knowledge on how they align with industry-standard processes. The secondary goal was to explore what they know, how they learn, and how they are committed to internal quality.

To get comparable information from different fields involved in software development we used two control groups. We asked the first control

group – at least one person from each target group – to evaluate our questions. They were given the survey with the explicitly stated aim of validating the expressions/sentences. Once the reported issues were corrected we created a second control group. This time participants were asked to fill in the survey on the web form it will appear later in. This was done in order to validate the corrections of earlier mentioned issues and to discover potential problems/technical difficulties with the layout of the survey. The results of the control groups were not included in the final survey.

To reach as many people as possible we created our anonymous survey with Google Forms using a minimum number of open ended questions. To track the knowledge of respondents we used questions with several predefined answers. To track the opinion of respondents we offered scales with five options. At some questions we asked for percentages of time spent with some activity.

We grouped the 47 survey questions (section C.1) into six sections:

1. "Generic information" established information, regarding the respondent's main role, task and size of their organization.

2. "Familiarity with different techniques" contained specific questions related to the four main targeted role groups to understand the actual knowledge of participants.

3. "Gaining new knowledge" collected information on how and from where participants gather new knowledge to improve their existing skills.

4. "Process and methodology related questions" assessed how many participants follow the industry-standard methods in their work.

5. "Anti-patterns" contained questions on how the participants are committed on the internal quality of their work.

6. "Static analysis and traceability" contained questions on static analysis tools, reviews, traceability issues.

Exploiting our social networks we contacted IT people from several companies (performing software development) and asked them to fill in and spread the survey within their companies (for example ERICSSON, NOKIA, LOGMEIN, NSN, SAP, NNG, PREZI, GE). We have also contacted several meetup[1] groups to let us advertise our survey on their site: TEST & TEA, HUNGARIAN C++ COMMUNITY, BUDAPEST DEVOPS MEETUP, FREELANCERS IN BUDAPEST. The survey was posted to the HUNGARIAN IT PROFESSIONALS group at `www.linkedin.com`. From the public forums we used `www.hup.hu`[2] and `www.prog.hu`[3].

We have also contacted the `Technical Writers` group of facebook.

[1] Communities organized on www.meetup.com

[2] Hungarian Unix Portal

[3] A web portal claiming to be the largest developer and programmer community in Hungary

## 6.2 Results regarding the roles

In total we received 456 responses from several professionals: 39 architects, 8 business operation supporters, 171 developers, 2 executive managers, 10 line managers, 3 manager of managers, 20 project managers, 2 self-employments, 28 team leaders, 28 technical writers and 145 testers.

To make the information processing easy we grouped the roles into four distinct groups: developers (210), testers (145), managers (71) and technical writers (28). At the end we decided to exclude responses from self-employed respondents. Their answers could not be merged into the other groups as they might do in their daily work all of the tasks of each group. At the same time we could not analyze their answers separately as that could have compromised their anonymity.

In order to be able to calculate statistics we mapped the "Not required – Required", "Never – Always", "Not concerned – Concerned" terms in the answers to the scale from one point to five.

### 6.2.1 Generic

86% of the respondents work for multi-national companies (85% of developers, 89% of testers, 81% of managers, 96% of technical writers). All but one technical writers responded to work for a multi-national company.

63% of the respondents work for companies having 1000+ employees. The ratio of testers is the highest in 501-1000 employee companies (52%), while the ratio of developers is the highest (70%) in companies employing 10 or fewer people (Fig. 6.1).

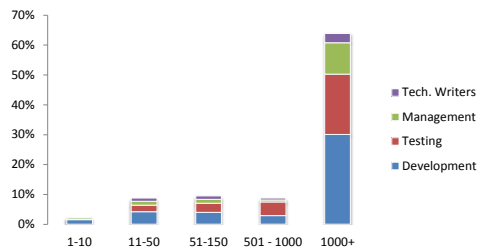



FIGURE 6.1: Company size distribution the employees are working for.

32% of the respondents work together with more than 30 people in their main project (Fig. 6.2). The second largest team size is 4-7. Most of the managers (47%) and testers (39%) work together with more than 30 people. Most of the developers (31%) work in projects of team size 4-7, just like technical writers.

51% of the respondents have less than 2 years of experience (29% have 3-5 years, 11% have 6-10 years, 7% have over 10 years of experience). We observed approximately the same ratio in all four groups (except that no technical writers reported to have 6-10 years of experience).

Figure 6.3 shows the tasks of people reflected to their role in 2014. Developers were developing systems (44% of all respondents), editing code (22%) and doing maintenance work (9%). Testers were testing (79%). Managers managed people (35%) and projects (20%). Technical writers wrote

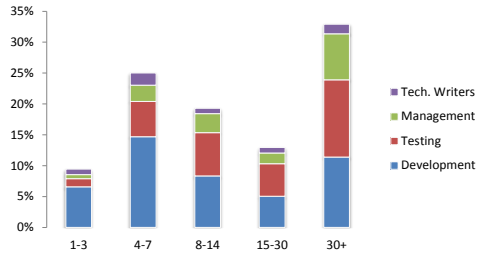



FIGURE 6.2: Group sizes the employees belong to in their main projects.

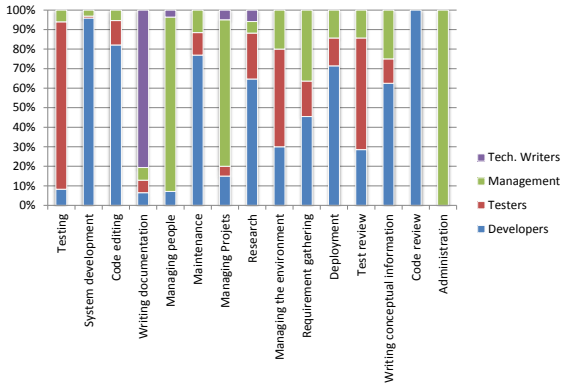



FIGURE 6.3: Tasks of the respondents.

documentation (89%). Only developers did code reviews (1%) as main task, the environment was mostly managed by testers (3%).

As most common additional responsibilities we recorded writing documentation (48%), testing (47%) and code review (43%). Test review and code editing took 4th and 5th place (37%) overall.

The most common secondary tasks were: for developers code review (67%) and testing (30%), for testers test review (67%) and writing documentation (53%), for managers managing people (42%) and administration (38%), for technical writers administration (39%) and product "research" (35%).

### 6.2.2 Familiarity with different patterns

Both developer and tester mindsets are very important in software development projects. While testing techniques are well known, development techniques rank as the least known.

The top three known design patterns are (Fig. 6.4): singleton (55%), iterator (52%) and factory (49%).

The top three known testing patterns are (Fig. 6.5): function testing (89%), use-case testing (69%) and review (64%).

The top three management methodologies are (Fig. 6.6): scrum (88%), agile (87%) and waterfall (82%).

The top three technical writer patterns are (Fig. 6.7): user documentation (64%), system documentation (59%) and review (38%).

In each experience level the ratio of people knowing any given design pattern is similar (Fig. 6.8).

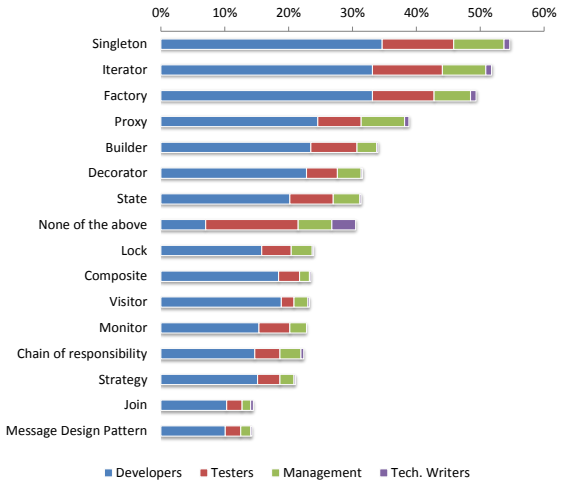



FIGURE 6.4: Knowledge of design patterns.

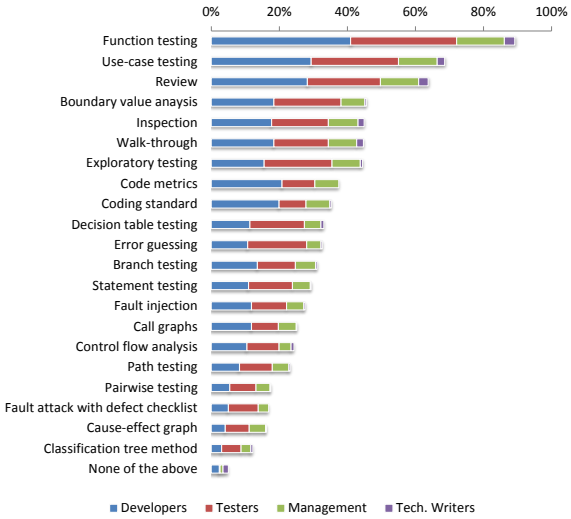



FIGURE 6.5: Knowledge of testing patterns.

Developers, testers and managers know approximately the same ratio of testing techniques, management techniques and technical writing techniques.

Technical writers know review, walk-through, inspection the most from testing techniques and scrum, agile and waterfall from management techniques. Technical writers have a balanced knowledge, more emphasis on analysis of audience, precise expressions proof-reading and less emphasis on user and system documentation.

Managers concentrate more on focus groups, documentation life-cycle management, and less on user testing and review.

Comparing all patterns we can see that the most known techniques are: Function testing (89%), Scrum (88%), User documentation (64%) and Singleton (55%).

Developer mindset was selected to be important (4-5 points) by all groups (93% developers, 61% testers, 65% managers and 46% technical writers). Testing mindset was selected to be important as well (4-5 points)

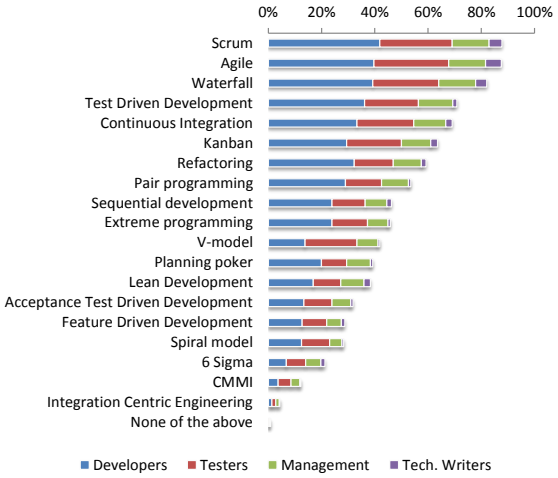



FIGURE 6.6: Knowledge of management methodologies/-patterns.

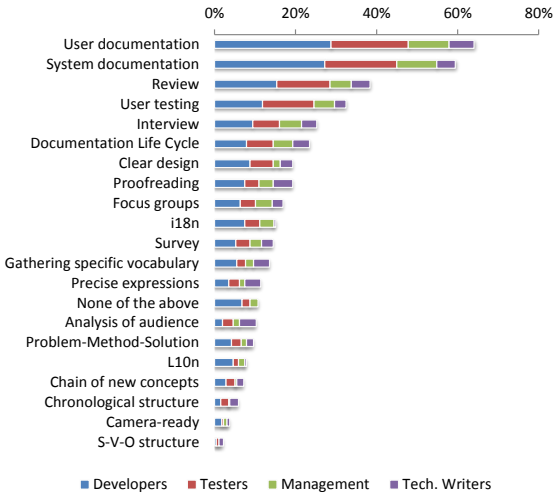



FIGURE 6.7: Knowledge of technical writing patterns.

by all groups (76% developers, 97% testers, 69% managers and 50% technical writers). Technical writer's mindset was selected to be important (4-5 points) mostly for technical writers (13% developers, 36% testers, 24% managers and 96% technical writers). Management mindset was selected to be important (4-5 points) mostly for managers (15% developers, 41% testers, 93% managers and 57% technical writers).

Altogether, the developer and tester mindsets were selected to be the most important in the software projects (Fig. 6.9). This points to an interesting observation: *testing mindset is reported to be important and testing techniques are well known, however, development techniques are the least known, but the mindset was still considered to be one of the most important.* Management mindset is considered to be only the 3rd on the importance level, still, some techniques are known by 30% more respondents than the most known development technique.

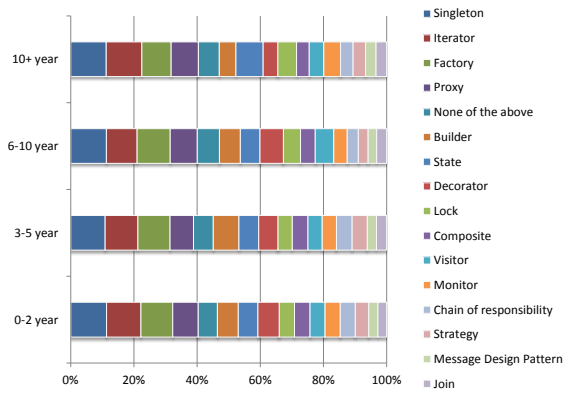



FIGURE 6.8: The most known software design patterns.

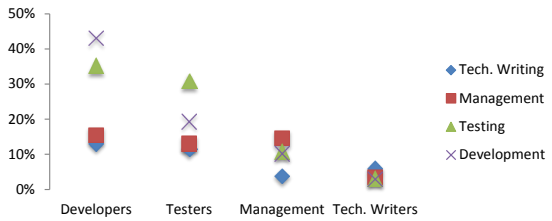



FIGURE 6.9: The importance of different mindsets.

### 6.2.3 Gaining new knowledge

The top three sources of new learning (Fig. 6.10) were: internet forums and blogs (82%), colleagues (79%) and books (65%). All 4 groups investigated show similar preferences. These results were repeated in the answer on what resources they have used in the previous year.

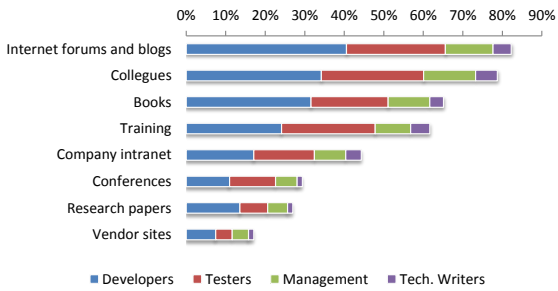



FIGURE 6.10: The main sources obtaining new knowledge.

Some additional sources for gaining new knowledge (that were not included but remarked in the questionnaire): meetups, online courses, self study.

We found (Fig. 6.11) that all role groups came by approximately the same ratio of knowledge through formal training (24%). However, the maximum ratio were very different: some developers could gain 100% of their knowledge in this way, while for technical writers the maximum was only 50%.

On-the-job training was most useful for managers (41% on average) and least useful for developers (30% on average). In this case the maximum ratio reported was 90% for technical writers, and 100% for all others.

Self study is the main source of knowledge for developers (44% on average), while technical writers use it the least (31% on average).

Trial and error is the source of 27-29% of knowledge for developers, testers and managers (on average). Some of them reported to gain 100% of their knowledge this way. Technical writers gain only 21% of their knowledge in this way (on average), and none of them reported to have gained more than 50%.

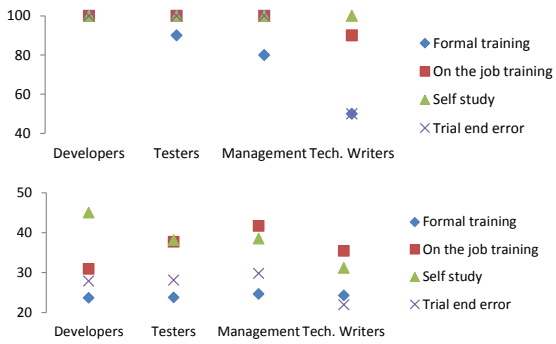



FIGURE 6.11: The maximum and average values of how knowledge is gained (in percentage).

Technical writers can rely the least on both formal trainings and learning by trial and error. They might get the same amount of knowledge from these methods, but they can get at most half of their experience in these ways (on average).

Formal trainings are less useful than trial and error based learning, and less people could claim to have learned everything on these ways.

### 6.2.4 Process and methodology

To be able to compare the different groups of people participating in software development projects we decided to check how strong their process and methodology orientation is versus an ad-hoc and intuitive practice in their daily work (we call this as "scientific" thinking). We asked whether (1) people are monitoring the world for new ideas and evaluate them critically before inserting into daily practices, (2) they are establishing hypotheses about the target before performing any change when the current situation is assessed, (3) people are able to detect if there is any flaw in the process planning, in the execution of the process or in the results, (4) the flaw is analyzed rigorously.

Results show (scores between 3 and 4) that at most companies it is somewhat important to work in a manner fulfilling strict processes and methodologies in order to see from where and to where tasks and people are heading, to understand where and in what position the work is standing.

*When we compared the main roles of the respondents based on their scientific thinking/methods we observed that respondents in development, testing and technical writing show similar values, while managers provided distributed answers* (Fig. 6.12 and Fig. 6.13). The average standard deviations for the process and methodology related questions (Fig. 6.13) in descending order: Q28 (1, 14), Q23 (1, 13), Q24 (1, 12), Q25 (1, 11), Q30 (1, 1), Q35 (1), Q34 (1), Q26 (1), Q32 (1), Q22 (1), Q33 (0, 99), Q31 (0, 99).

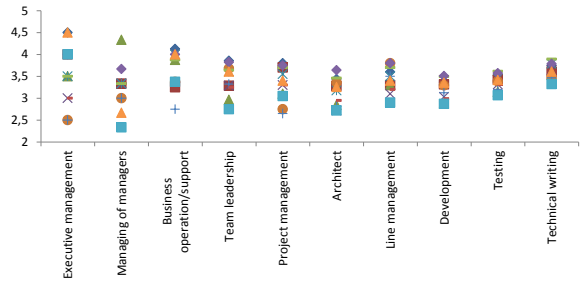



FIGURE 6.12: Process and methodology importance depending on different roles (1: least, 5: most, on average)

Checking the correlation coefficients between the average answers given by people having different roles revealed the following:

- *The highest correlation could be found between developers and testers:* 0.93 *on average.*
- Developers and architects way of thinking had the second largest correlation coefficient: 0.90.
- The team leadership way of thinking is correlated with (1) development: 0.89, (2) testing: 0.88, (3) line management: 0.85, (4) architect: 0.84.
- The architect way of thinking is correlated with: (1) development: 0.90, (2) testing: 0.89, (3) team leadership: 0.85, (4) line management: 0.82.
- We also observed a correlation of 0.80 between technical writing and testing mindsets.

All other correlation coefficients were below 0.80. The process and methodology orientation of the management (including executive management, managing of managers, business operation/support, project and line management) has little in common with each other and with other roles. In the technical writer's thinking they are the closest to testing (0.80) and development (0.78).

Respondents reported (Fig. 6.14) that in their company the newest technologies/methodologies are frequently monitored and evaluated (from never to always: 2%, 15%, 26%, 40%, 16%). Mostly managers and technical writers responded with confirmative values (∼70%), but even most of the developers and testers perceived that their organizations perform these tasks (∼50% confirmative answers).

We had similar distribution of answers for the question of how extensively new technologies are tested before introducing them into the organizations's life (from never to always: 5%, 20%, 26%, 33%, 19%). We found that developers, managers and technical writers gave 4-5 marks in ∼50%, while testers in ∼60% of the cases (Fig. 6.15). Technical writers found their tools best tested before introduction.

The answers for the question "how often testable hypotheses are established before work starts?" were in the middle of the frequency range (Fig.

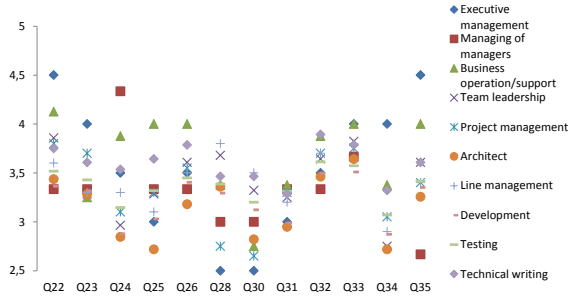



FIGURE 6.13: Answers for process and methodology related questions (1: least, 5: most, on average). Q22: technology awareness, Q23: technology introduction, Q24: technology/methodology pilot, Q25: process improvement, Q26, Q28, Q30: process modification, Q31: process introduction, Q32, Q33: process definition, Q34: process performance, Q35: process redesign. (The full questions can be found in the Appendix)

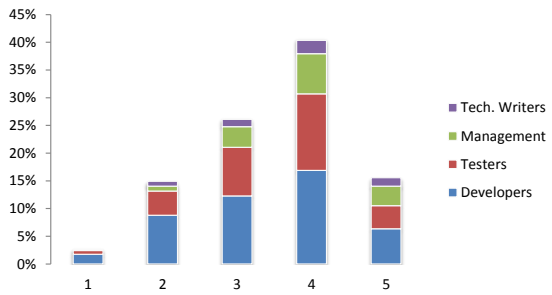



FIGURE 6.14: Monitoring and evaluating the newest technologies/methodologies in the company (number of respondent, 1: never, 5: always).

6.16). In this case the different groups had very different perceptions. Developers gave the fewest high values and technical writers the most.

When an activity is not done as specified respondents mostly follow a defined process to improve it (Fig. 6.17). Developers rate their improvement processes the weekest, while technical writers the best.

When the outcome is defective despite all activities done as specified respondents mostly modify their processes. Again, developers gave the lowest values for the frequency of their process modification procedures, while technical writers gave the highest values.

Approximately half of the respondents in all groups reported the ability to detect when someone is idle for long and then follow a defined process to modify or reassign activities. Respondents reported to have been idle 9-15% of their time in 2014 independently from their roles. The maximum idle ratio was almost twice longer for developers and testers, and almost one and half times longer for managers than for technical writers.

Approximately 40% of the developers, testers and managers reported high confirmatory values (4-5 points) for being able to detect if someone is overloaded and then being able to follow a defined process in order to modify or reassign activities. The average ratio of being overloaded in 2014

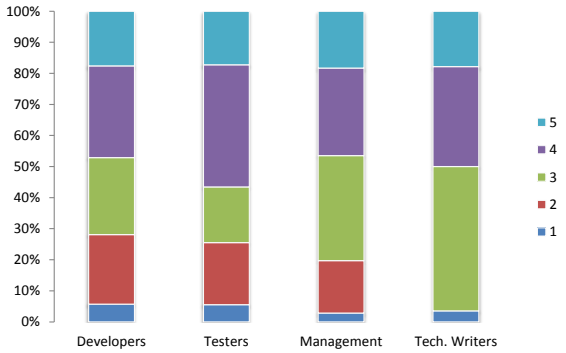



FIGURE 6.15: When a new piece of technology/methodology is available, extensive testing is performed before introducing it into the organisation's life (1: never, 5: always).

was 24% for developers, 28% for testers and 31% for managers and technical writers. The maximum reported ratio of being overloaded was very high in all groups.

Only ∼30% of developers and technical writers are able to redesign their processes (4-5 points) if they find that a non-specific activity is needed compared to the ∼45% of testers and managers.

In all groups, the respondents were able to detect easily what the next activity is in their processes being actually performed. High values (4-5 points) were given by ∼55% of the developers, ∼60% of the testers, ∼64% of the managers and ∼68% of the technical writers. In all groups only ∼15% of the respondents gave scores below 3.

We observed the same ratio for determining who has to perform the next activity in the process.

Only ∼30% of the developers, testers and managers check the current state of affairs rigorously before making a change, compared to 45% of technical writers. Only 5% of the respondents reported that they always asses the current state of affairs before making a change.

When the results are not the ones expected ∼50% of the developers, testers and technical writers check how the change was done and what effects it might had, compared to the 60% of managers.

When we looked at how people in the different roles rate their processes and methodologies (Fig. 6.13) we get some interesting insights into how much and where their thinking (perception of their processes) differs. Based on the average values of each role for each question (that fall outside the 3.2 - 4 range):

- Executive Managers believe they are monitoring new technologies (4.5) and carefully testing them before integration (4). The current state is assessed before making a change (4) and if a change has a different effect than expected the reason is checked (4.5). At the same time they believe they are the least likely to identify if someone is idle (2.5) or overloaded (2.5); to find the reason for non-specific activities (3) or improving in case of wrong execution (3).

- Managers of managers believe they set up hypotheses before work starts (4.3), but are least likely to check the reason if the result of a change is different from the expected (2.6).


FIGURE 6.16: When a new activity/artifact is defined then sets of hypotheses are estabilished that can be tested before work starts (1: never, 5: always).

- Business operation/support believe they asses the current state rigorously (4), know clearly what the next activity in the process is (3.8) and who has to carry out the next activity (4). They try to improve after a bad outcome (4) and modify processes (4). At the same time they are bad at telling who is overloaded (2.75) and testing new technology before introducing it to the processes (3.3).

- Team leaders believe they are bad at finding out who has to carry out the next activity (2.75) and establishing a testable hypotheses before work starts (3).

- Project Managers find it hard to identify idle (2.75) and overloaded persons (2.65). They also don't believe they create testable hypotheses before starting the work (3.1), or assessing current state of affair with rigor (3).

- Architects generally give scores between 2.9 and 3.5. They don't believe to have an improvement process to follow when something goes wrong (2.7), or to assess the current state before making changes (2.7). They also don't believe they create good hypotheses before starting the work (2.8), or find out why a non-specific activity is needed (2.9), or telling if someone is overloaded (2.8).

- Line managers believe they have good processes for telling who is idle (3.8), what the next activity is (3.7) and who has to carry it out (3.78). They don't believe they are assessing the current state before a change (2.9) or follow a process to improve (3.1).

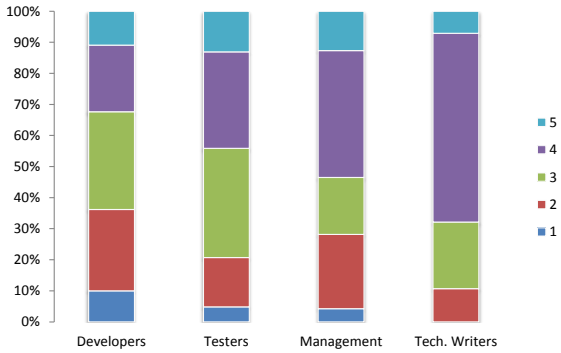



FIGURE 6.17: When an activity is not done as specified a defined process is followed in order to improve it (1: never, 5: always).

- Developers generally give scores between 3.1 and 3.4. They don't believe they assess the current state (2.87) and establish a hypotheses (2.87) before starting the work. They also don't believe that, when something is not done as specified or some extra activity is need, they follow a process to improve (3) and redesign their processes (3).

- Testers generally give scores between 3.2 and 3.5. They don't believe they assess the current state (3) and establish a hypotheses (3.1) before starting the work. They also don't believe that their team is able to identify overloaded people (3.2).

- Technical writers generally give scores between 3.5 and 4. They believe it is clear what the next activity is (3.9), who has to carry it out (3.78) and when they find defective outcome, in spite of doing everything right, they modify their processes (3.78). They least believe they can find out why some non-specific activity is need (3.28) or assess the current state of affair with rigor (3.32).

### 6.2.5 Anti-patterns

Although most companies support the improvement of internal quality, most respondents have never heard of or are not concerned about anti-patterns.

We have described anti-patterns as "an anti-pattern is a common response to a recurring problem that is usually ineffective and risks being highly counterproductive" in the survey.

35% of the respondents answered to have never heard of them, and 20% to have heard of anti-patterns but not sure what they are. 15% know them, but are not concerned. Only 25% reported trying to avoid them, and 2% reported a strong understanding. Anti-patterns are most understood by developers, and least by testers (Fig. 6.18).

When checked the question in more detail, we got that 51% of the architects tries to avoid them, 87% of business operations/support have never heard of them or are not sure what they are. 26% of the developers have never heard of them, 19% are not sure what they are, 19% are not concerned, 33% try to avoid them, but only 2% have strong knowledge and use tools to detect and remove them. Line, executive and manager's managers

have balanced knowledge (half of them are familiar with anti patterns on some level, half of them not). 75% of project managers have never heard of them, are not sure what they are, or are not concerned. Only 12% of the testers know and try to avoid them and only 1% uses tools for detection and removal.

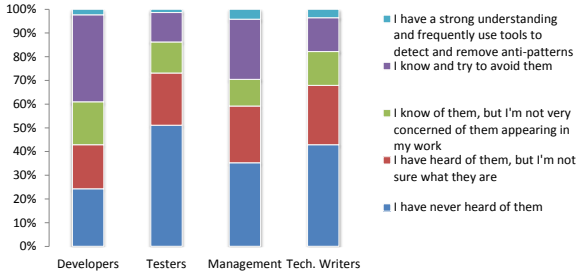



FIGURE 6.18: Familiarity with design anti-patterns.

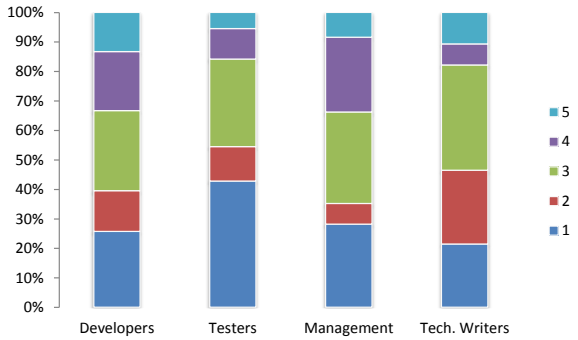



FIGURE 6.19: The severity of taking anti-patterns into consideration (1: least, 5: most).

When asked how concerned respondents are about anti-patterns in their product, 31% of them reported to be not concerned and 30% to be mildly concerned. In all role groups at least 20% of the respondents were not concerned at all and only 5-15% were concerned (Fig. 6.19). Developers (13%) and technical writers (10%) being the most concerned.

The result means that:

- at least 60% of the respondents in all groups are supported by his organization to improve the internal quality of their products (Fig. 6.20). The ratio is the best (65%) for technical writers.
- at least 40% of the respondents either have pre-planned sessions and work lists for internal quality improvements or correct such issues immediately when they notice them.
- less than 6% have reported to have no organizational support for internal quality improvements.
- less than 7% have reported to have no process for internal quality improvements.

In 2014 most respondents produced low quality results in order to satisfy short term needs 1-5 times (35% 1-2 times, 29% 3-5 times). There were

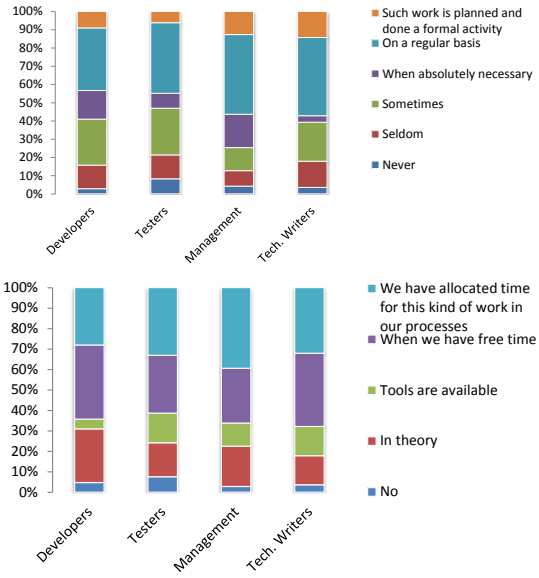



FIGURE 6.20: (above) The abundance of working on existing products in order to improve their internal quality. (below) The necessity of working on existing products to improve their internal quality supported by your organization.

68 respondents who did not need to give up on quality (Fig. 6.21), while 11% produced low quality 10+ times. The ratio of no compromises was best among technical writers (21%), followed by developers (18%), testers (13%) and managers (7%).

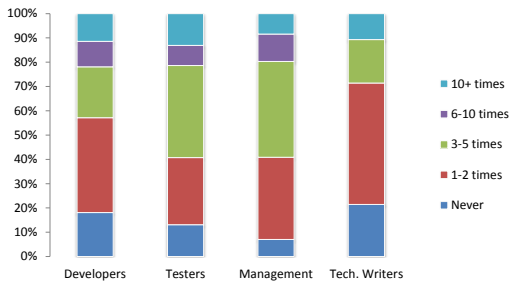



FIGURE 6.21: The frequency of producing low quality solutions in order to satisfy short term needs in 2014.

### 6.2.6 Static analysis and traceability

Our further analysis shows that most of the found issues are traced back to the early stages of processes and are controlled by static tools, manual code reviews and direct contact to customers.

According to respondents most issues can be traced back to code writing, concept/system design and requirement collection (Fig. 6.22). Regarding the role groups we had similar rates except that technical writers found task management as a source of problems principally and placed less emphasis on code writing.

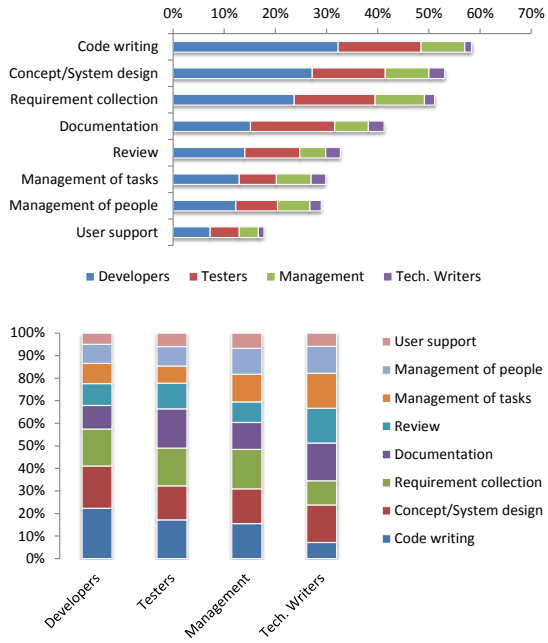



FIGURE 6.22: Processes to which the found issues were traced back in 2014.

Both technical writers and testers placed the most emphasis on the available documentation as the source of problem solving. Most organizations apply tools to statically check adherence to coding standards and to measure metrics (Fig. 6.23). At this point we observed a clear difference in the roles: developers, testers and managers take static analysis tool supports by approximately the same percentage in their work, but technical writers reported to be less supported in checking coding standards and measuring metrics. They also reported the highest ratio of not being supported by static analysis tools.

We asked furthermore whether manual code reviews are used in internally developed products: 73% answered yes (81% of developers, 67% of testers, 74% of managers and only 39% of technical writers, Fig. 6.24).

The average time manual reviews took, was 51 minutes for testers and technical writers, 56 minutes for managers and 58 minutes for developers. The maximum time spent with manual reviews was 8 hours for managers and technical writers, 16 hours for developers, and testers could spend up to 24 hours.

By our measurements 40% of the respondents selected to have no direct contact with their users (Fig. 6.25). After direct email (51%) this option received the second most votes.

Some respondents mentioned that customer support and issue tracking tools are "the" direct contact to users.

The question "How do you judge if a specification is out-of-date?" was offered to the respondents in order to describe the situation with their own words. 30% of the respondents gave any answer. 4% categorized as "do not care", 2% answered to check the version number of the appropriate document, 1.9% decided to verify the date of the last modification, 2.5% would ask for help to decide. 1% of the respondents answered that their processes

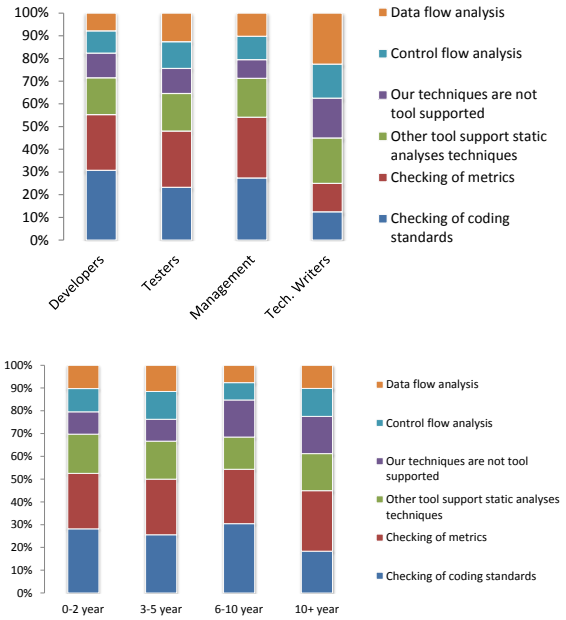



FIGURE 6.23: The application of tool supported static analysis techniques (above roles:, below: experiences).

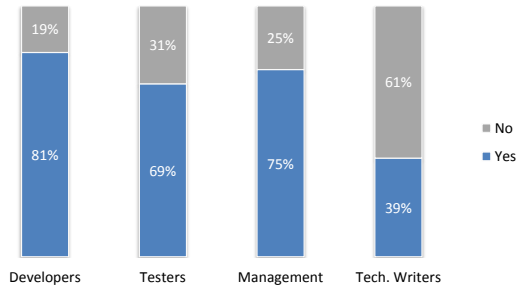



FIGURE 6.24: Manual code reviews for internally developed products.

make out of date documents impossible. 2% would compare it to the code or existing features. Other 1% of the respondents mentioned some mechanisms or tools that are able to check the validity of the specification before work starts. Rest of the responses either did not understand the question, or could not be categorized in larger groups. For example: "working on prototype means documents are always outdated", "have not happened yet", "by my standards", "too bad".

## 6.3 Results through the size of the company

In this section we analyse the different mindsets through the size of the company.

*We experienced that bigger companies have more career options, more experienced people, better processes, better quality validations and better on the job training instead of reliance on self-study. Bigger companies use their resources more efficiently, without overloading them more, without indirectly forcing them to produce lower quality.*


FIGURE 6.25: Types of direct contacts with customers.

In all companies with less than 1000 employees we found only 1 employee with 10+ years of experience, while 1000+ employee companies employ 6%.

As the size of companies grows, more job roles appear (1-10: 5; 11-50: 7; 51-150: 7; 151-1000: 8; 1000+: 9).

The larger the company is, the more developer mindset is demonstrable. In companies with 1-10 employees three times more people selected 5 (most important) for the importance of the developer mindset than 1 (least important). In 1000+ companies the ratio is twenty three. The same is true for the testing mindset with multipliers in the range of 2-150. In the case of the management mindset the multiplier is 2-3 in all company size ranges. Technical writer mindset is the most important in 51-150 employee companies, but on absolute scale they receive the most 5-s in 1000+ employee companies (10%).

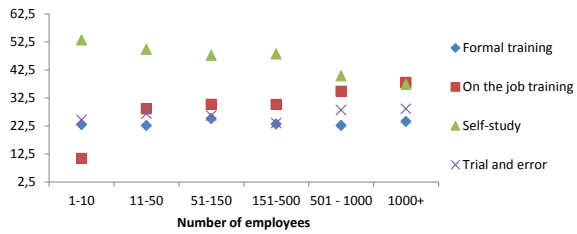



FIGURE 6.26: The average ratio of knowledge gaining methods depending on the size of the company (in percentage).

We observed (Fig. 6.26) that the average ratio of the on-the-job training gained knowledge is larger in bigger companies. While at smaller companies employees get only ∼11% of their knowledge through on-the-job training, at 1000+ companies this ratio is 38%. For self-study we observed the opposite trend: as the size of the company increases the average ratio of knowledge gained through self-study decreases from 53% to 37%. The size of the company had no significant effect on the average ratio of trial and error and formal training based knowledge retrieval.

Regarding the methodology related questions we found that the size of the company has a noticeable impact on the quality: almost all investigated characteristics were slightly increased/improved with the size (Fig. 6.27)).

The size of the company has a negative linear correlation with the number of people being idle in the previous year. The average idle time was 23% in the smallest companies while 12% at the largest. We found no correlation between the company size and the number of people overloaded:

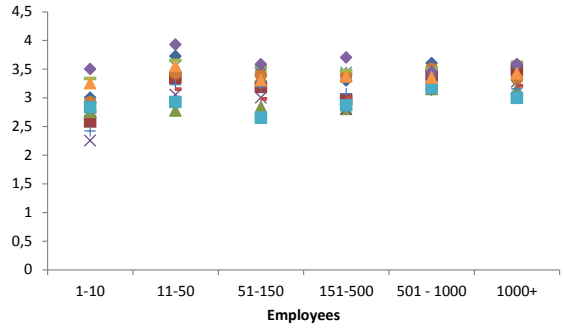



FIGURE 6.27: The average points reported for each method-/thinking related question, shown by the size of the company.

in companies of 151-500 employees the average overloaded time ratio was 18%, while at other companies 27-32%.

In 1000+ employee companies 24% of respondents knows and tries to remove anti-patterns and 2% uses tools for this. In all other company size ranges there were only a few respondents per size range applying tools for detecting and removing anti-patterns. The ratio of those who know and try to avoid anti-patterns is ∼33% in companies below 500 employees (17% at 501-1000 and 23% at 1000+ companies).

Independently of the size of the company there are two times as many respondents correcting internal quality when there is an issue than those who can correct them during development and testing without planned quality improvement sessions.

In company sizes above 150 employees ∼6-10% of employees produced low quality solutions 10+ times to satisfy short term needs. In smaller companies this ratio was 13-25%. In companies below 1000 employees only a few respondents answered producing quality without compromises. In contrast, in 1000+ employee companies it is ∼16%.

With the size of the company the existence of manual code reviews performed were growing as expected: 33% at 1-10 sized companies, ∼60% between 10-500 sized companies and 80% at 500+ companies. The duration of the manual code reviews were: 20 minutes at 1-10 sized companies, 45 minutes at 11-50 sized companies, 35 minutes at 51-150 sized companies and 65 minutes above (in average).

## 6.4 Results through experience levels

There are various ways to consider experiences. One of our most surprising observation was that *spending more time at the same working place changes the way of thinking only a little*.

We were interested in where the experienced employees are working. The respondents having 10+ years of experiences consist of ∼7% of all employees, ∼14% have 6-10 years, ∼26% have 3-5 years and ∼53% of the employees have less than or equal to two years of experiences. Figure 6.28 shows the distribution of experiences in various team sizes.

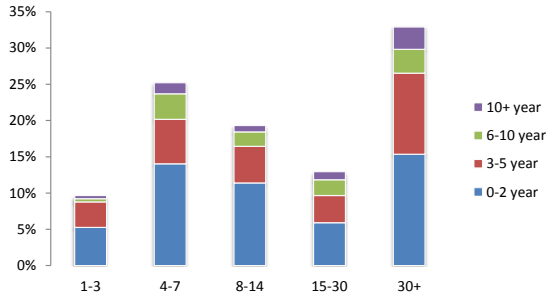



FIGURE 6.28: The distribution of experiences in various team sizes.

The importance of the mindsets is similar in all experience ranges (Fig. 6.29). We measured that technical writer mindset gets more important with experience, the management mindset drops back in the 10+ years experience group. The developer and tester mindsets did not change significantly with the experiences.

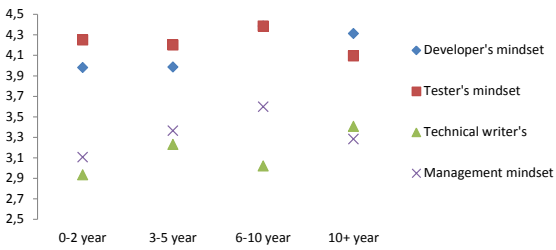



FIGURE 6.29: The average importance of mindsets by experience groups (in percentage).

In all experience groups the average amount of knowledge, used in work, acquired through on-the-job training/self-study/trial and error is approximately constant, while the average amount of knowledge gained through formal training drops from ∼24% to ∼16% at 10+ years of experience (Fig. 6.30).

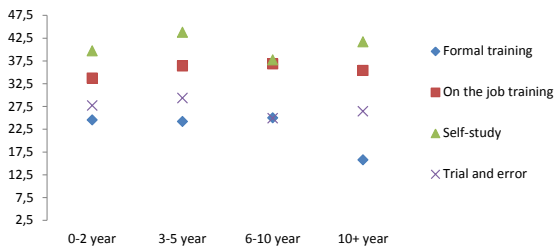



FIGURE 6.30: Acquiring knowledge by experience groups (in percentage).

We observed as well that the knowledge of design patterns, testing and management techniques known does not depend on the respondents working experiences. However, technical writer techniques knowledge changes with working experience: the importance of user documentation, system documentation and reviews rises until 10 years of experience. After 10 years of experience the importance of user and system documentation no

longer increases: reviews and user testing fall back. Proofreading shows an opposite trend: its usage drops back with experience, but after 10 years of experience it becomes the 3rd most known technique.

We examined the experience through the thinking/method related questions. We found that the answers for almost all questions were approximately the same in all experience groups. Until 6-10 years of experience the most improved properties were: understanding who has to carry out the next step (17% increase) and checking how the change was done when the result is not as expected (11% increase). Some properties even fall back: monitoring and evaluating the newest technologies/methodologies (18% drop), detecting if someone is idle for too long (11% drop) and learning why a non-specific activity is needed (18% drop).

The biggest progression happens between 6 and 10 years of experience: monitoring and evaluating the newest techniques/methodologies (44% increase), extensive testing before introduction (31% increase), learning why a non-specific activity is needed (30% increase).

The average amount of being idle drops from 12-14% to 4% when reaching 10+ years of experience. The average amount of being overloaded slowly grows from 25% to 31%.

In all experience groups the ratio of respondents using tools to detect and remove anti-patterns was under 2%. The ratio of respondents who know about anti-patterns and try to avoid them was between 20-30% in all experience groups.

From all experience range the employees traced back the issues to the same sources, with the same ratio with one exception: only 1 respondents with 10+ years of experience selected user support as the source of problems. He also placed more emphasis on reviews than others with the same or less experience.

After 10 years of experience all employees are working some time on internal quality improvements. The ratio of regularly improving internal quality was the highest in this group: 50%. At the same time 43% of them is improving internal quality in his free time, while only ∼30% of people with less experience reported the same..

With the amount of experience the average time spent at manual reviews rises from 40 to 75 minutes.

# Chapter 7

# Summary

In this thesis I aimed at analyzing the internal quality of TTCN-3 based test systems.

To create a stable base I connected the TTCN-3 language to the international software quality standards ISO-9126 and ISO-25010 by defining and classifying 86 code smells. In order to be able to measure the quality of the test systems I designed and developed a tool by which I found several internal quality issues in both industrial and standardized TTCN-3 test suites. I analyzed and assessed the costs of correcting the found issues of the defined code smell items. I estimated that most of these might need thousands of man-hours to correct.

I analyzed the architectural properties of TTCN-3 based test systems. I extended our tool with a layered visualization layout and architecture extraction possibilities. I surveyed that this layout the asked industrial test system architects found useful. I analyzed standardized and industrial test suits by which I was able to show that the examined TTCN-3 test systems contain issues on architectural level and our visualization solution makes it easier to detect these issues comparing to other available solutions.

I analyzed how the internal quality of test systems change during their evolution. I measured two test systems over a five years period. I concluded that changing the development processes, project leaders, team and technical leaders, introducing continuous integration and automated quality checks did not cause significant difference in the number of code smell instances present. I observed predictable tendencies, just like Lehman's law predicted, showing similarity with the evolution of software systems.

I run a survey to understand the human side of writing quality tests and code. I showed that from human aspects regarding the internal quality a test project is very similar to a software project. This hints at a kind of "convergence" between testing and development which others (e.g. [126, 127, 128]) have already noticed. I experienced that bigger companies have more career options, more experienced people, better processes, better quality validations and better on the job training instead of reliance on self-study. Bigger companies use their resources more efficiently, without overloading them more, without indirectly forcing them to produce lower quality. I also found that most companies support the improvement of internal quality, but most respondents have never heard of or are not concerned about anti-patterns.

# Összefoglaló

A doktori dolgozat TTCN-3 -ban írt tesztrendszerek kódminőségének vizsgálatáról szól.

Az elemzésekhez először a TTCN-3 nyelvhez kapcsolódó gyanús kódmintákat határoztam meg (code smells), majd ezeket az ISO-9126 és ISO-25010 szoftverminőség szabványoknak megfelelően osztályoztuk. A minőség méréséhez eszközt terveztem és fejlesztettem, aminek a segítségével ipari és sztenderd TTCN-3 testsorozatok kódminőségét vizsgáltam. Elemeztem és megbecsültem továbbá a talált nem-megfelelőségek refaktorálásához szükséges ráfordítások költségét.

Megvizsgáltam a TTCN-3 alapú tesztrendszerek strukturális tulajdonságait, rétegzett elrendezésű megjelenítő eljárást készítettem és implementáltunk. Módszeremet az ipari tesztrendszer tervezők is hasznosnak találták. Vizsgálatom eredményei közül kiemelhetőek az alábbiak: (1) a szabadon elérhető tesztsorozatok közül több is tartalmaz projekttől független modulokat, körkörös importokat modul és könyvtár szinten egyaránt; (2) a modulok közötti kimenő import kapcsolatok logaritmikus görbével, míg a bemenő import kapcsolatok hatványgörbével közelíthetőek; (3) a vizsgált gráfok átmérője logaritmikus függvénye a projektben található modulok számának.

Ezután a tesztsorozatok időbeli változását vizsgáltam két tesztrendszer ötéves fejlődésén keresztül. A vizsgálatok során azt találtam, hogy a fejlesztési módszertan, a projektvezetők, a csapat és a technikai vezetők változása, valamint a CI és az automatizált minőségellenőrzés bevezetése nem járt számot tevő hatással a gyanús kódminták számára nézve. A Lehman törvényekkel analóg módon – a szoftver-rendszerek fejlődéséhez hasonlóan – a teszrendszerek esetére is érvényes törvényszerűségeket sikerült kimutatnom.

A minőségi tesztek és kódok írása emberi vonatkozásainak feltérképezéséhez kérdőíves felmérést végeztem. A szakmai gondolkodásra/módszerekre vonatkozó kérdéseimre a fejlesztők és a tesztelők adták a leghasonlóbb válaszokat. Ez egyfajta "konvergenciára" utal a tesztelés és fejlesztés között, amit mások (pl. [126, 127, 128]) már megsejtettek. Megállapítható, hogy bár a legtöbb vállalatnál támogatják a termékek belső minőségének javítását, a válaszadók jelentős része mégsem hallott rossz mintákról (anti-patterns), vagy nem tartja ezek jelenlétét a tesztekben, kódokban aggályosnak.

# Bibliography


[1] B.W. Boehm, P.N. Papaccio: *Understanding and Controlling Software Costs*, 1988, IEEE Transactions on Software Engineering, **14**/10 (1988) pp. 1462–1477.

[2] A. Avram: *IDC Study: How Many Software Developers Are Out There?*, 2014, https://www.infoq.com/news/2014/01/IDC-software-developers, last visited: January, 2017.

[3] G. Tassey: *The Economic Impacts of Inadequate Infrastructure for Software Testing*, 2002, Final report, Prepared by RTI for the National Institute of Standards and Technology (NIST), https://www.nist.gov/sites/default/files/documents/director/planning/report02-3.pdf, last visited: January, 2017.

[4] European Telecommunications Standards Institute, Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language Version 1.0.10, http://www.etsi.org/deliver/etsi_es/201800_201899/20187301/01.00.10_50/es_20187301v010010m.pdf, last visited: January, 2017.

[5] European Telecommunications Standards Institute, Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language Version 4.5.1, http://www.etsi.org/deliver/etsi_es/201800_201899/20187301/04.05.01_60/es_20187301v040501p.pdf, last visited: January, 2017.

[6] T. Kamada, S. Kawai: *An algorithm for drawing general undirected graphs*, Information Processing Letters, Volume 31, Issue 1, 1989, pp. 7–15.

DOI:10.1016/0020-0190(89)90102-6

[7] T.M.J. Fruchterman and E.M. Reingold: *Graph drawing by force-directed placement*, Software-Practice & Experience, 21/11, 1991, pp. 1129–1164.

DOI:10.1002/spe.4380211102

[8] S. Hachul, M. Junger: *Large-Graph Layout Algorithms at Work: An Experimental Study*, Journal of Graph Algorithms and Applications, Vol. 11, No. 2, 2007, pp. 345–369.

DOI: 10.7155/jgaa.00150

[9] TITAN, https://projects.eclipse.org/proposals/titan, last visited: January, 2017.

[10] W. Cunningham: *The wycash portfolio management system*, In Proceedings of OOPSLA '92 Addendum to the proceedings on Object-oriented programming systems, languages, and applications (Addendum), ACM, 1992, pp. 29–30.

DOI: 10.1145/157710.157715

[11] CAST, *Technical debt estimation*, http://www.castsoftware.com/researchlabs/technical-debt-estimation, last visited: January, 2017.

[12] A. Kyte: *Measure and manage your IT debt*, Gartner Inc., 2010, https://www.gartner.com/doc/1419325/measure-manage-it-debt, last visited: January, 2017.

[13] I. Griffith, D. Reimanis, C. Izurieta, Z. Codabux, A. Deo, B. Williams: *The Correspondence between Software Quality Models and Technical Debt Estimation Approaches*, In 6th International Workshop on Managing Technical Debt (MTD), 2014, pp. 19–26.

DOI: 10.1109/MTD.2014.13

[14] J. Holvitie, V. Leppanen, S. Hyrynsalmi: *Technical Debt and the Effect of Agile Software Development Practices on It – An Industry Practitioner Survey*, In 6th International Workshop on Managing Technical Debt (MTD), 2014, pp. 35–42.

DOI: 10.1109/MTD.2014.8

[15] T.S. Mendes, M.A.F. Farias, M.l Mendonca, H.F. Soares, M. Kalinowski, and R.O. Spinola: *Impacts of agile requirements documentation debt on software projects: a retrospective study*, In Proceedings of the 31st Annual ACM Symposium on Applied Computing (SAC '16), ACM, New York, USA, 2016, pp. 1290–1295.

DOI: http://dx.doi.org/10.1145/2851613.2851761

[16] N. Ramasubbu, C.F. Kemerer: *Managing Technical Debt in Enterprise Software Packages*, In IEEE Transactions on Software Engineering, Volume 40, Issue 8, 2014, pp. 758–772.

ISSN: 0098-5589, DOI: 10.1109/TSE.2014.2327027

[17] J. Ho, G. Ruhe: *When-to-release decisions in consideration of technical debt*, In 6th International Workshop on Managing Technical Debt (MTD), 2014, pp. 31–35.

DOI: 10.1109/MTD.2014.10

[18] Z. Li, P. Avgeroiu, P. Liang: *A systematic mapping study on technical debt and its management*, Journal of Systems and Software, Volume 101, 2014, pp. 193–220.

DOI:10.1016/j.jss.2014.12.027

[19] M. Fowler: *Refactoring: Improving the Design of Existing Code*, 1999, Addison-Wesley Longman Publishing Co. Inc., Boston, USA.

ISBN-10: 0-201-48567-2, ISBN-13: 978-0201485677

[20] E.V. Emden, L. Moonen: *Java Quality Assurance by Detecting Code Smells*, *Proceedings of the Ninth Working Conference on Reverse Engineering* (WCRE'02), IEEE Computer Society, Washington DC, USA, 2002, pp. 97–106.

[21] N. Moha, Y.G. Gueheneuc, L. Duchien, and A.-F. Le Meur: *Decor: A method for the specification and detection of code and design smells*, 2010, IEEE Transactions on Software Engineering, Volume 36/1, pp. 20–36.

ISSN: 0098-5589, DOI: 10.1109/TSE.2009.50

[22] H. Neukirchen, M. Bisanz: *Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites*, 2007, Proceedings of the 19th IFIP International Conference on Testing of Communicating Systems and 7th International Workshop on Formal Approaches to Testing of Software (TestCom/FATES 2007),

Tallinn, Estonia. Lecture Notes in Computer Science (LNCS) 4581, Springer, 2007, pp. 228–243.

ISBN: 978-3-540-73065-1, DOI: 10.1007/978-3-540-73066-8_16

[23] F. Khomh, M.D. Penta, Y.G. Guhéneuc: *An exploratory study of the impact of code smells on software change-proneness*, 2009, Proceedings of the 16th Working Conference on Reverse Engineering, WCRE '09, IEEE Computer Society, Washington DC, USA, 2009, pp. 75–84.

[24] S. Olbrich, D. Cruzes, V.R. Basili, N. Zazworka: *The evolution and impact of code smells: A case study of two open source systems*, 2009, Proceedings of the 3rd International Symposium on Empirical Software Engineering and Measurement, ESEM '09, IEEE Computer Society, Washington DC, USA, 2009, pp. 390–400.

[25] B. Geppert, A. Mockus, F. Robler: *Refactoring for changeability: a way to go?*, 11th IEEE International Software Metrics Symposium (METRICS'05), 2005, pp. 10–13.

DOI: 10.1109/METRICS.2005.40

[26] M. Abbes, F. Khomh, Y.G. Gueheneuc, G. Antoniol: *An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension*, In Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR '11), IEEE Computer Society, Washington DC, USA, 2011, pp. 181–190.

DOI: http://dx.doi.org/10.1109/CSMR.2011.24

[27] I.S. Deligiannis, I. Stamelos, L. Angelis, M. Roumeliotis, M.J. Shepperd: *A controlled experiment investigation of an object-oriented design heuristic for maintainability*, Journal of Systems and Software, 72, 2004, pp. 129–143.

[28] Cs. Faragó, P. Hegedűs and R. Ferenc: *Code Ownership: Impact on Maintainability*, in Computational Science and Its Applications (ICCSA 2015), Springer, 2015, pp. 3–19.

DOI: 10.1007/978-3-319-21413-9_1

[29] Cs. Faragó, P. Hegedűs and R. Ferenc: *The Impact of Version Control Operations on the Quality Change of the Source Code*, in Computational Science and Its Applications (ICCSA 2014), Springer, 2014, pp. 353–369.

DOI: 10.1007/978-3-319-09156-3_26

[30] Cs. Faragó, P. Hegedűs, G. Ladányi and R. Ferenc: *Impact of Version History Metrics on Maintainability*, in Proceedings of the 8th Intenational Conference on Advanced Software Engineering and Its Application (ASEA), 2015, IEEE Computer Society, pp. 30–35.

DOI:10.1109/ASEA.2015.14

[31] R. Moser, P. Abrahamsson, W. Pedrycz, A. Sillitti, D. Succi: *A Case Study on the Impact of Refactoring on Quality and Productivity in an Agile Team*, In *Balancing Agility and Formalism in Software Engineering*, Springer, 2008, pp. 252–266.

ISBN: 978-3-540-85278-0, DOI: 10.1007/978-3-540-85279-7_20

[32] E. Ammerlaan, W. Veninga and A. Zaidman: *Old habits die hard: Why refactoring for understandability does not give immediate benefits*, IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Montreal, QC, 2015, pp. 504–507.

DOI: 10.1109/SANER.2015.7081865

[33] M. Zhang, T. Hall, N. Baddoo: *Code Bad Smells: a review of current knowledge*, Journal of Software Maintenance and Evolution: Research and Practice **23**/3, 2011, pp. 179–202.

[34] A. Monden, D. Nakae, T. Kamiya, S. Sato, K. Matsumoto: *Software quality analysis by code clones in industrial legacy software*, Symposium on Software Metrics, 2002, pp. 87–94.

[35] W. Li, R. Shatnawi: *An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution*, Systems and Software, **80**/7, 2007, pp. 1120–1128.

[36] D.I.K. Sjøberg, A. Yamashita, B. Anda, A. Mockus, and T. Dyba: *Quantifying the effect of code smells on maintenance effort*, IEEE Trans. Softw. Eng., 39(8), 2013, pp. 1144–1156.

[37] A. Yamashita: *Assessing the capability of code smells to explain maintenance problems: an empirical study combining quantitative and qualitative data*, Empirical Softw. Engg., 19/4 (August 2014), 2014, pp. 1111–1143.

[38] A. Yamashita: *Assessing the Capability of Code Smells to Support Software Maintainability Assessments: Empirical Inquiry and Methodological Approach*, 2012, Doctoral Thesis, University of Oslo.

[39] A.v. Deursen, L. Moonen, A.v.d. Bergh, and G. Kok: *Refactoring test code*, Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes (XP2001), University of Cagliari, 2001, pp. 92–95.

[40] B. Zeiss, D. Vega, I. Schieferdecker, H. Neukirchen, and J. Grabowski: *Applying the ISO 9126 Quality Model to Test Specifications - Exemplified for TTCN-3 Test Specifictions*, Software Engineering, Lecture notes in Informatics (LNI) **105**, Gesellschaft für Informatik, Köllen Verlag, Bonn, 2007, pp. 231–242.

[41] H. Neukirchen, B. Zeiss, J. Grabovszki: *An Approach to Quality Engineering of TTCN-3 Test Specifications*, 2008, International Journal on Software Tools for Technology Transfer (STTT), **10**/4, (ISSN 1433-2779), Springer, pp. 309–326.

DOI: 10.1007/s10009-008-0075-0

[42] ISO/IEC 9126:1991: *ISO Standard for Software Engineering – Product Quality* Revised by ISO/IEC 9126–1:2001

[43] ISO/IEC 25010:2011: *ISO Systems and Software Engineering - Systems and Software Quality Requirements and Evaluation (SQuaRE) – System and Software Quality Models*

[44] ISO/IEC 15504-5:2012: *Information technology – Process assessment* http://www.iso.org/iso/catalogue_detail.htm?csnumber=60555, last visited: January 2017.

[45] *CMMI institute*: http://cmmiinstitute.com/, last visited: January 2017.

[46] R.v. Solingen, E. Berghout: *The goal/question/metric method, a practical method for quality improvement of software development*, 1999, McGraw-Hill.

ISBN: 007-709553-7

[47] *The Personal Software Process (PSP) Body of Knowledge*, Version 2.0; Special Report; CMU/SEI-2009-SR-018.

[48] W.S. Humphrey: *The Team Software Process*, Technical Report, CMU/SEI-2000-TR-023, ESC-TR-2000-023, 2000.

[49] A. Bánsághi, B.G. Ézsiás, A. Kovács, A., Tátrai: *Source Code Scanners in Software Quality Management and Connections to International Standards*, Annales Univ. Sci. Budapest Sect. Comp., 37, 2012, pp. 81–92.

[50] Test Process Improvement, https://www.sogeti.com/solutions/testing/tpi/, last visited: January 2017.

[51] Test Maturity Model Integration, https://www.tmmi.org, last visited: January 2017.

[52] Systematic Test and Evaluation Process, http://flylib.com/books/en/2.174.1.11/1/, last visited: January 2017.

[53] Critical Testing Process: Plan, Prepare, Perform, Perfect, http://dl.acm.org/citation.cfm?id=861686, last visited: January 2017.

[54] ISO/IEC 9646: *Information technology - Open Systems Interconnection - Conformance testing methodology and framework*, http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=17473, last visited: January 2017.

[55] L. Bass, P. Clements, R. Kazman: *Software Architecture In Practice*, 1998, Addison Wesley.

[56] D. Budgen, *Software Design*, Pearson Education, 2003.

ISBN: 0-201-72219-4

[57] U. van Heesch, P. Avgeriou: *Mature Architecting – a survey about the Reasoning Process of Professional Architects*, Software Architecture (WICSA), 9th Working IEEE/IFIP Conference on, Boulder, CO, 2011, pp. 260–269.

DOI: 10.1109/WICSA.2011.42

[58] A. Tang, P. Avgeriou, A. Jansen, R.L. Capilla, M.A. Babar: *A comparative study of architecture knowledge management tools*, J. Syst. Softw. 83/3, 2010, pp. 352–370,

DOI: http://dx.doi.org/10.1016/j.jss.2009.08.032

[59] P. Kruchen: *Games Architects Play*, 2011, http://www.cs.rug.nl/~matthias/pages/workshop_april_18_2011/slides_kruchten.pdf, last visited: January, 2017.

[60] W. Stacy, J. MacMillan: *Cognitive bias in software engineering*, Commun. ACM., Vol. 38, 1995, pp. 57–63.

[61] A. Tang: *Software designers, are you biased?*, In Proceedings of the 6th International Workshop on SHAring and Reusing Architectural Knowledge (SHARK '11)., ACM, New York, USA, 2011, pp. 1–8.

DOI: http://dx.doi.org/10.1145/1988676.1988678

[62] U. van Heesch, P. Avgeriou, A. Tang: *Does decision documentation help junior designers rationalize their decisions? A comparative multiple-case study*, J. Syst. Soft. 86/6, 2013, pp. 1545–1565.

DOI: http://dx.doi.org/10.1016/j.jss.2013.01.057

[63] F.A. Fontana, S. Maggioni: *Metrics and Antipatterns for Software Quality Evaluation*, In Proceedings of the IEEE 34th Software Engineering Workshop (SEW '11), IEEE Computer Society, Washington DC, USA, 2011, pp. 48–56.

DOI: http://dx.doi.org/10.1109/SEW.2011.13

[64] D. Binkley, N. Gold, M. Harman, Z. Li, K. Mahdavi and J. Wegener: *Dependence Anti Patterns*, In Automated Software Engineering Workshops, ASE Workshops 2008, 23rd IEEE/ACM International Conference on L'Aquila, 2008, pp. 25–34.

DOI: 10.1109/ASEW.2008.4686318

[65] M. Feikas, D. Ratiu, E. Jurgens: *The loss of Architectural Knowledge during System Evolution: An Industrial Study*, In IEEE 17th International Conference on Program Comprehension, 2009, pp. 188–197

DOI: 10.1109/ICPC.2009.5090042

[66] T.D. Oyetoyan, D.S. Cruzes, R. Conradi: *Criticality of Defects in Cyclic Dependent Components*, In 13th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM), 2013, pp. 21–30.

DOI: 10.1109/SCAM.2013.6648180

[67] T. Zimmermann, N. Nagappan: *Predicting Subsytem Failures using Dependency Graph Complexities*, In 18th IEEE International Symposium on Software Reliability (ISSRE), 2007, pp. 227–236

DOI: 10.1109/ISSRE.2007.19

[68] A. Schroter, T. Zimmermann, A. Zeller: *Predicting Component Failures at Design Time*, In International Symposium on Empirical Software Engineering, 2006, pp. 18–27

DOI: 10.1145/1159733.1159739

[69] H. Melton, E. Tempero: *An empirical study of cycles among classes in java*, In Empirical Software Engineering, Vol. 12, Issue 4, 2007, pp. 389-415

DOI: 10.1007/s10664-006-9033-1

[70] J. Dietrich, C. McCartin, E. Tempero, S.M.A. Shah: *Barriers to Modularity – An empirical study to assess the potential for modularisation of Java programs*, In Proceedings 6th International Conference on the Quality of Software Architectures, 2010, pp. 135–150.

DOI: 10.1007/978-3-642-13821-8_11

[71] P. Caserta, O. Zendra: *Visualization of the Static Aspects of Software: A Survey*, In IEEE transaction on Visualization and Computer Graphics, Volume 17, Issue 7, 2011, pp. 913–933.

DOI: 10.1109/TVCG.2010.110

[72] M. Shahin, P. Liang, M.A. Babar: *A systematic review of software architecture visualization techniques*, J. Syst. Software, Volume 94, 2014, pp. 161–185.

DOI: 10.1016/j.jss.2014.03.071

[73] S. Reiss: *The Paradox of Software Visualization*, In Proceedings of the 3rd IEEE International Workshop on Visualizing for Understanding and Analysis (VISSOFT), 2005, pp. 59–63.

DOI: 10.1109/VISSOF.2005.1684306

[74] A. Kuhn, D.D. Erni, O. Nierstrasz: *Embedding spatial software visualization in the IDE: an exploratory study*, In Proceedings of the 5th international symposium on Software visualization (SOFTVIS '10), 2010, pp. 113–122.

DOI:10.1145/1879211.1879229

[75] R. Albert, H. Jeong, A.L. Barabási: *Error and attack tolerance of complex networks*, Nature, Vol. 406/6794, 2000, pp. 378–382.

[76] G. Zipf: *Psycho-Biology of Languages*, 1935, Houghtton-Mifflin, Boston.

[77] H. Jeong, B. Tombor, R. Albert, Z.N. Oltvai, A.L. & Barabási: *The large-scale organization of metabolic networks*, Nature, Vol. 407, 2000, pp. 651–654.

[78] A. L. Barabási: *Linked – The New Science of Networks*, Perseus Press, New York, 2002.

[79] D. Hyland-Wood, D. Carrington, S. Kaplan: *Scale-Free Nature of Java Software Package, Class and Method Collaboration Graphs*, In the 5th International Symposium on Empirical Software Engineering, September 21-22, Rio de Janeiro, Brazil, 2006.

[80] A. Potanin, J. Noble, M. Frean, R. Biddle: *Scale-free geometry in OO programs*, Communications of the ACM, Vol. 48, Issue 5, 2005, pp. 99–103.

[81] A.P. de Muora, Y.C. Lai, A.E. Motter: *Signatures of small-world and scale-free properties in large computer programs*, Physical Review, E 68(1-2), 017102, 2003, pp. 171021-171024.

[82] M.M. Lehman and J.F. Ramil: *Towards a theory of software evolution – and its practical impact* (working paper), Invited Talk, Proceedings Intl. Symposium on Principles of Software Evolution, ISPSE, 2000, pp. 2–11.

[83] M.M. Lehman and J.F. Ramil: *Rules and tools for software evolution planning and management*, Ann. Software Eng., 11(1), 2001, pp. 15–44.

[84] M.M. Lehman and J.F. Ramil: *Evolution in software and related areas*, Proceedings of the 4th International Workshop on Principles of Software Evolution, IWPSE '01, ACM, New York, USA, 2001, pp. 1–16.

[85] M.J. Lawrence: *An examination of evolution dynamics*, Proceedings of the 6th International Conference on Software Engineering, ICSE '82, Los Alamitos, CA, USA, IEEE Computer Society Press, 1982, pp. 188–196.

[86] C. Izurieta and J. Bieman: *The evolution of freebsd and linux*, Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering, ISESE '06, ACM, New York, USA, 2006, pp. 204–211.

[87] W.M. Turski: *The reference model for smooth growth of software systems revisited*, IEEE Trans. Software Eng., 28(8), 2002, pp. 814–815.

[88] J.F. Ramil, D.I. Cortazar and T. Mens: *What Does It Take to Develop a Million Lines of Open Source Code*, in Open Source Ecosystems: Divers Communities Interacting, OSS 2009, IFIP Advances in Information and Communication Technology, vol 299, Springer, Berlin, 2009, pp 170–184

DOI: 10.1007/978-3-642-02032-2_16

[89] M.M. Lehman, J.F. Ramil, and D.E. Perry: *On evidence supporting the feast hypothesis and the laws of software evolution*, Proceedings of the 5th International Symposium on Software Metrics, METRICS '98, Washington DC, USA, 1998, IEEE Computer Society, pp. 84–99.

[90] C.F. Kemerer and S. Slaughter: *An empirical approach to studying software evolution*, IEEE Trans. Software Eng., 25(4), 1999, pp. 493–509.

[91] M.M. Lehman: *Feast/2 final report – grant number gr/m44101*, 2001.

[92] A. Israeli and D.G. Feitelson: *The linux kernel as a case study in software evolution*, J. Syst. Software, 83(3), 2010, pp. 485–501.

[93] K. Johari and A. Kaur: *Effect of software evolution on software metrics: An open source case study*, SIGSOFT Software Eng., Notes, 36(5), 2011, pp. 1–8.

[94] A. Chatzigeorgiou and A. Manakos: *Investigating the evolution of bad smells in object-oriented code*, Proceedings of the 2010 Seventh International Conference on the Quality of Information and Communications Technology, QUATIC '10, Washington DC, USA, IEEE Computer Society. 2010, pp. 106–115,

[95] D.L. Parnas: *Software aging*, Proceedings of the 16th International Conference on Software Engineering, ICSE '94, Los Alamitos, CA, USA, IEEE Computer Society Press, 1994, pp. 279–287,

[96] R. Peters and A. Zaidman: *Evaluating the lifespan of code smells using software repository mining*, Proceedings of the 16th European Conference on Software Maintenance and Reengineering, CSMR '12, Washington DC, USA, IEEE Computer Society, 2012, pp. 411–416.

[97] A. Zaidman, B. Rompaey, A. Deursen, and S. Demeyer: *Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining*, Empirical Software Eng., 16(3), 2011, pp. 325–364.

[98] A. Koenig: *Patterns and antipatterns*, In The patterns handbooks, Linda Rising (Ed.). Cambridge University Press, New York, USA, 1998, pp. 383–389.

ISBN:0-521-64818-1

[99] J. Carr: *TDD anti-patterns*, http://blog.james-carr.org/2006/11/03/tdd-anti-patterns/, last visited: January 2017.

[100] A. Scott: *Introducing the software testing ice-cream cone (anti-pattern)*,

http://watirmelon.com/2012/01/31/introducing-the-software-testing-ice-cream-cone/, last visited: January 2017.

[101] N. Juristo, A.M. Moreno, and S. Vegas: *A Survey on Testing Technique Empirical Studies: How Limited is our Knowledge*, In Proceedings of the 2002 International Symposium on Empirical Software Engineering (ISESE '02), IEEE Computer Society, 2002, pp. 161–172.

DOI: 10.1109/ISESE.2002.1166935

[102] A.M.J. Hass: *Guide to Advanced Software Testing*, Artech House, 2008.

ISBN-13: 978-1596932852

[103] I. Stamelos, R. Charikleia, T. Poramen, E. Berki: *Software Project Management Anti-patterns in Students' Projects*, http://www.sis.uta.fi/~tp54752/pub/Anti-patternsinStudentsProjects.pdf, last visited: January 2017.

[104] G.J. Alread, C.T. Brusaw, W.E. Oliu: *Handbook of Technical Writing*, Bedford St. Martin's, 2011.

ISBN-13: 978-0312679453

[105] H. Femmer, D.M. Fernández, S.N. Wagner, S. Eder: *Rapid quality assurance with Requirements Smells*, Journal of Systems and Software, Volume 123, 2017, 190–213.

ISSN 0164-1212, DOI: 10.1016/j.jss.2016.02.047

[106] A. Yamashita and L. Moonen: *Do developers care about code smells? An exploratory survey*, Proceddings of the 20th Working Conference on Conference: Reverse Engineering, IEEE Computer Society, 2013, pp. 242–251.

DOI: 10.1109/WCRE.2013.6671299

[107] A. Yamashita and L. Moonen: *Do code smells reflect important maintainability aspects?*, Proceedings of the IEEE International Conference on Software Maintenance, ICSM '12, Washington DC, USA, IEEE Computer Society, 2012, pp. 306–315.

ISSN: 1063-6773, DOI: 10.1109/ICSM.2012.6405287

[108] A. Yamashita and L. Moonen: *Exploring the impact of inter-smell relations on software maintainability: An empirical study*, Proceedings of the International Conference on Software Engineering, ICSE '13, Piscataway, NJ, USA, 2013, IEEE Computer Society Press, 2013, pp. 682–691.

[109] g. Calikli, A.Bener: *Empirical analysis of factors affecting confirmation bias levels of software engineers*, Software Quality Journal, Volume 23 Issue 4, 2015

DOI: 10.1007/s11219-014-9250-6

[110] State of Testing Survey report:

http://www.practitest.com/wpcontent/uploads/2015/07/State_of_Testing_Survey_2015.pdf, last visited: January 2017.

[111] ISTQB Worldwide Software Testing Practices Report 2015-2016,

http://www.istqb.org/references/surveys/istqb-worldwide-software-testing-practices-report-2015-2016.html, last visited: January 2017.

[112] *PMD*, http://pmd.sourceforge.net, last visited: January 2017.

[113] *FxCop*, http://msdn.microsoft.com, last visited: January 2017.

[114] *Checkstyle*, http://checkstyle.sourceforger.net, last visited: January 2017.

[115] *FindBugs*, http://findbugs.sourceforge.net, last visited: January 2017.

[116] G. Meszaros: *xUnit Test Patterns: Refactoring Test Code*, Addison-Wesley,

ISBN-10: 0131495054, ISBN-13: 978-0131495050

[117] *TRex*, http://www.trex.informatik.uni-goettingen.de/trac, last visited: January 2017.

[118] EUROPEAN TELECOMMUNICATIONS STANDARDS INSTITUTE, ETSI EG 201 015 V2.1.1, Methods for Testing and Specification (MTS), Standards engineering process; A Handbook of validation methods, http://www.etsi.org/deliver/etsi_eg/201000_201099/201015/02.01.01_60/eg_201015v020101p.pdf, last visited: January 2017.

[119] L. Helmer: *Analysis of the future: The Delphi method*, RAND Corporation, 1967, http://www.rand.org/pubs/papers/P3558.html, last visited: January 2017.

[120] R. Cohen and D. Hevlin: *Scale-Free Networks are Ultrasmall*, Physical Review Letters, Vol. 90/5, 058701, 2003. https://doi.org/10.1103/PhysRevLett.90.058701, last visited: January 2017.

[121] Java Universal Network/Graph Framework,
http://jung.sourceforge.net/, last visited: January 2017.

[122] J. Lakos: *Large-scale C++ software design*, Addison-Wesley Professional, 1996, pp. 312–324.

[123] S. Mancordis, B.S. Mitchell, C. Rorres, Y. Chen, E.R. Ganser: *Using Automatic Clustering to Produce High-Level System Organizations of Source Code*, In Proceedings of the 6th International Workshop on Program Comprehension (IWPC '98), IEEE Computer Society, Washington, DC, USA, 1998, pp. 45–52.
ISBN: 0-8186-8560-3, DOI: 10.1109/WPC.1998.693283

[124] M.E. Conway: *How do committees invent?*, Datamation, 14(5), 1968, pp. 28–31.

[125] M.M. Lehman: *Laws of software evolution revisited*, Proceedings of the 5th European Workshop on Software Process Technology, EWSPT '96, Springer, 1996, pp. 108–124.

[126] Soasta, *Could developers be the future of software testing?* http://www.soasta.com/blog/could-developers-be-the-future-of-software-testing/, last visited: January 2017.

[127] K. Katdare: *Career In Software Testing Vs. Software Development*,
http://www.crazyengineers.com/threads/career-in-software-testing-vs-software-development.67131/, last visited: January 2017.

[128] S. Rowe: *Hiring Great Testers – How Important Is Testing Affinity?*,
http://blogs.msdn.com/b/steverowe/archive/2007/02/13/hiring-great-testers-how-important-is-testing-affinity.aspx, last visited: January 2017.

[129] A. Yamashita and L. Moonen: *To what extent can maintenance problems be predicted by code smell detection? - an empirical study*, Inf. Software Techn., 55/12, 2013, pp. 2223–2242.

[130] N. Sangal, E. Jordan, V. Sinha, D.Jackson: *Using dependency models to manage complex software architecture*, In Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '05), 2005, pp. 167–176.
DOI:10.1145/1094811.1094824

[131] I. Macia, J. Garcia, D. Popescu, A. Garcia, N. Medvidovic, and A. von Staa, *Are automatically-detected code anomalies relevant to architectural modularity?: An exploratory analysis of evolving systems*, Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development, AOSD '12, ACM, New York, 2012, pp. 167–178.

[132] I. Stamelos: *Software project management anti-patterns*, Journal of Systems and Software, Elsevier, Vol. 83, 2010, 52–59.
DOI: 10.1016/j.jss.2009.09.016

[133] W. Brown, R. Malveau, H. McCormick, T. Mowbray: *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*, Wiley, 1998.
ISBN: 978-0-471-19713-3

## Own papers, conference talks, posters

[134] K. Szabados: *Structural Analysis of Large TTCN-3 Projects*, In Proceeding Of Testing of Software and Communication Systems, 21st IFIP WG 6.1 International Conference, TESTCOM 2009 and 9th International Workshop, FATES 2009, Eindhoven, The Netherlands, November 2-4, Lecture Notes in Computer Science: Testing of Software and Communication Systems, Springer, 2009, pp. 241–246.

ISBN: 978-3-642-05030-5, DOI: 10.1007/978-3-642-05031-2_19

[135] K. Szabados and A. Kovács: *Test software quality issues and connections to international standards*, Acta Universitatis Sapientiae, Informatica, 5/1, 2013, pp. 77–102.

DOI: 10.2478/ausi-2014-0006

[136] K. Szabados and A. Kovács: *Advanced TTCN-3 Test Suite validation with Titan*, In Proceedings of the 9th International Conference on Applied Informatics, Vol. 2, 2014, pp. 273–281.

DOI: 10.14794/ICAI.9.2014.2.273

[137] K. Szabados and A. Kovács, *Technical debt of standardized test software*, IEEE 7th International Workshop on Managing Technical Debt (MTD), Bremen, 2015, pp. 57–60.

DOI: 10.1109/MTD.2015.7332626

[138] K. Szabados and A. Kovács, *Up-to-date list of code smells*, http://compalg.inf.elte.hu/~attila/TestingAtScale.htm, last visited: January, 2017.

[139] K. Szabados, A. Kovács, G. Jenei and D. Góbor: *Titanium: Visualization of TTCN-3 system architecture*, IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR), Cluj-Napoca, Romania, 2016, pp. 7–11.

DOI: 10.1109/AQTR.2016.7501275

[140] K. Szabados and A. Kovács: *Knowledge and mindset in software development – how developers, testers, technical writers and managers differ – a survey*, 11th Joint Conference on Mathematics and Computer Science (MACS), Eger, Hungary, 2016. State: accepted for publication.

[141] K. Szabados and A. Kovács: *Internal quality evolution of a large test system – an industrial study*, Acta Universitatis Sapientiae, Informatica, 8/2, 2016, 216–240.

[142] K. Szabados: *Creating an efficient and incremental IDE for TTCN-3*, 10th Joint Conference on Mathematics and Computer Science, Cluj-Napoca, In Studia Universitatis Babes-Bolyai, Informatica, Volume LX, Number 1, 2015, pp. 5–18.

[143] K. Szabados and A. Kovács: *Developing and Testing at Large Scale*, 5th Annual International Conference of the Hungarian Software Testing Forum (HUSTEF), Budapest, Hungary, 2015. (Talk)

[144] K. Szabados: *Thinking/mindset of testers is the closest to that of developers*, 6th International Conference of the Hungarian Software Testing Forum (HUSTEF), Budapest, Hungary, 2016. (Poster)

[145] K. Szabados, Gy. Réthy: *Test Software Quality Through Software Metrics*, 1st User Conference on Advanced Automated Testing (UCAAT 2013), Paris, 2013. (Poster)

[146] K. Szabados, A. Kovács: *Test systems, software systems. Is there a difference?*, 3rd User Conference on Advanced Automated Testing (UCAAT 2015), ETSI, Sophia Antipolis, 2015. (Talk)

# Appendix A

# TTCN-3

TTCN-3[1] is a high level standardized language designed for testing. Mostly used for functional testing (conformance testing, function testing, integration, verification, end-to-end and network integration testing) and performance testing. TTCN-3 can be used (1) to test reactive systems via: message based communication, (2) API based and analog interfaces and systems.

The language is governed by a strict, internationally accepted specification. Each language construct, allowed by the syntax and semantics of the standard, has a well specified behavior. Tests written in TTCN-3 can be transfered to other vendor's tools without modification. Some standards of reactive systems (for example communication protocols) offer their specifications together with a set of tests written in TTCN-3. This provides an easy and automated way for tool vendors and users to check the conformance of the implementation.

TTCN-3 offers platform independent abstract data types (see listing A.1). There is no value range restriction for integers, no precision restriction for floats, and no length restriction for string types. String types are differentiated based on their contents (bitstring, hexstring, octetstring, charstring, universal charstring). Creating new types is supported by building structured types with fields (record, set) or by list of an element type (record of, set of). It is also possible to create new types with restriction (for example length restriction on strings). This rich type / data constructs can easily be extended by importing other data types / schema (ASN.1[2], IDL[3], XSD[4] and JSON[5]) without need for manual conversion.

The templates of TTCN-3 merge the notions of test data and test data matching into one concept (see listing A.2). This enables the specification of expected responses in a concise way. Matching rules can be for example: single value ("Budapest"), list of alternatives ("Monday", "Tuesday"), range (1 .. 5), ordered and unordered lists of values, sub- and supersets of unordered values, string patterns (pattern"* chapter"), permutations of values. When declaring templates for structured data types, these matching rules can be declared for each field and element individually or for the whole template. Checking whether a data value matches to the template is as easy as "match(value, templateValue)". Other constructs offer additional functionality, e.g. "*.receive(templateValue) -> value" activates only if a value matching to the provided template is received, in which case the value of the message is saved in "value" for further processing.

[1] Test and Test Control Notation 3
[2] Abstract Syntax Notation One
[3] Interface Definition Language
[4] XML Schema Definition
[5] JavaScript Object Notation

LISTING A.1: data types example

```
var boolean v_boolean:=true;
const integer c_i:= 123456789101112131415;
const float c_f1:=1E2;
const float c_f2:=100.0;
var bitstring v_bits:='01101'B;
var charstring v_chars:="ABCD";
var hexstring v_hexs := '01A'H;
var octetstring v_octs:='0BF2'O;
var universal charstring v_uchars := "F" & char(0, 0, 0, 65)

type record recordOper_trecord {
        integer x1 optional,
        float x2 };
type record of octetstring recordOper_trecof;
type set recordOper_tset {
        integer x1,
        float x2 optional };
type set of charstring recordOper_tsetof;

type integer templateInt_subtype (0..1457664);
type record length (3)
  of record length (3)
  of record length (3) of integer threeD;
```

LISTING A.2: templates example

```
template integer t_i := 123456789101112131415
var template float vt_f := (1.0 .. 2.0);
template mycstr t_mycstr := pattern "ab" & "cd";

template templateCharstr_rec templateCharstr_tList :={
 x1:="00AA", //specific value
 x2:=("01AA","01AB","11AC"), // value list
 x3:=complement ("11","0A","1BC0"), // complement list
 x4:=? length(2..4), //any string with a length of 2 to 4
 x5:= pattern "10*" //any string matching the pattern
  };
```

LISTING A.3: Example for receiving message

```
testcase tc_HelloWorld() runs on MTCType system MTCType
{
  timer TL_T := 15.0;
  map(mtc:MyPCO_PT, system:MyPCO_PT);
  MyPCO_PT.send("Hello, world!");
  TL_T.start;
  alt { //branching based on events
    [] MyPCO_PT.receive("Hello, TTCN-3!") {
        TL_T.stop;
        setverdict(pass);//receiving the right message
       }
    [] TL_T.timeout {
        setverdict(inconc); // the test timed out
       }
    [] MyPCO_PT.receive {
        TL_T.stop; // some other message was received
        setverdict(fail);
       }
  }
}
```

TTCN-3 can also be viewed as a "C -like" procedural language with testing specific extensions. The usual programming language features (function, if, while, for, etc. ) are extended with other constructs needed for testing: test cases as standalone constructs, sending/receiving messages, invoking remote procedures and checking the content of the received data structures (messages/results/exceptions), alternative behaviors depending on the response of the tested entity, handling timers and timeouts, verdict assignment and tracking, logging of events (see listing A.3) are all built in.

Creating distributed test cases and test execution logic is easy as well. A TTCN- 3 test may consist of several parallel test components which are distributed on a set of physical machines, able to work in tandem to test all interfaces of the tested system, or able to create high load. Test components, communication ports to the tested entity and to other test components are defined in TTCN-3. The number of test component instances and their connections are controlled from the code of the test case dynamically using various language features (see listing A.4). Deploying and controlling the test component also happens in an abstract and platform independent way. The user does not need to work with the implementation details. It is the tools responsibility to utilize the available pool of machines, possibly running on different operating systems.

TTCN-3 is also independent from the test environment. The user needs only to define abstract messages exchanged between the test system and test tested entity. Message encoding (serialization), decoding (deserialization), handling of connections and transport layers are done by the tools.

TTCN-3 also offers to control the test case execution logic and dynamic test selection from within the the TTCN-3 code itself (see listing A.5). Module parameters allow for the user to leave data open in the source code and provide the actual values at execution time (IP addresses, IDs, passwords, etc...)

LISTING A.4: multiple components example

```
testcase commMessageValue() runs on commMessage_comp2 {
var commMessage_comp1 comp[5];
var integer xxint;
for (var integer i:=0; i<5; i:=i+1)
{ log(i);
  comp[i]:=commMessage_comp1.create;//creating component
  comp[i].start(commMessage_behav1(i));//start remote behavior
  connect(self:Port2[i],comp[i]:Port1);//connect to component
  xxint:=5;
  Port2[i].send(xxint);//send message on port
  Port2[i].receive(integer:?) -> value xxint;//receive response
  if (xxint==5+i) {setverdict(pass)}
    else {setverdict(fail)};
}
for (i:=0; i<5; i:=i+1) {comp[i].stop};//stop the components
};
```

LISTING A.5: execution control example

```
control {
  for(var integer i := 0; i < 10; i := i+1)
  {
    execute(parameterised_testcase(i));
  }
  execute(transferTest());
  execute(tc_runsonself());
 }
```

In the foreseeable future the worlds of telecommunication and the Internet will converge together as fast as never before (IoT, autonomous driving, etc.), the systems to be tested will become more dynamic and complex in their nature. TTCN-3 contains all the important features to specify test procedures for functional, conformance, interoperability, load and scalability tests, its test-specific features are unique compared to traditional script-based testing languages, and above all, technology-independent. Hence it seems to be an appropriate choice for the above mentioned challenges.

# Appendix B

# Code smells

## B.1 Defined smells

In the following we enumerate the Code Smells defined or found applicable to TTCN-3:

1. FIXME tags: Developer markings of severe incorrect or missing features.
2. TODO tags: Developer markings of incorrect or missing features.
3. Circular importation: The import relation of modules forms at least one loop.
4. Duplicated code: Very similar code exists in more than one location.
5. Similar functions: Several functions differing only in literal values.
6. Mergeable templates: Similar data structures, that could be merged into a single parameterized one.
7. Long statement blocks: A block of statements that has grown too large.
8. Too many parameters: A long list of formal parameters.
9. Excessively short identifiers: The name of an identifier is too short to reflect it's functions.
10. Excessively long identifier: The name of an identifier is too long.
11. Divergent naming: The identifier breaks the naming conventions.
12. "Private" group: Public definitions categorized in a group called "private".
13. Internal comments: Internal comments indicate too complicated code.
14. Missing comments: All methods should be commented.
15. Type in method name: The return type's name is redundant in the method name.
16. Module in method name: The containing module is mentioned in the method name.
17. Visibility embedded in name: Visibility rules evaluated by user.
18. Incomplete literals: Some fields of literals and constants are left uninitialized/unbound.
19. Initialize with constant: Structured value declared without initial value.

20. Dummy fields in constants: Field always overridden, should be left unbound.
21. Goto detection: Goto is considered to break structured programming rules.
22. Unnecessary imports: Module importations that are unnecessary.
23. Unused global definitions: Some global definitions are not used.
24. Unused local definitions: Some local definitions are not used.
25. Unnecessary operations: Operations never executed.
26. Unchecked module parameter: The module parameter is used before being checked.
27. Push definition to component: Functions running on a component define the same local variable.
28. Pull definition to local: A component member is only used in a few functions.
29. Unused return value: The result or error handling of the function call is missing.
30. Unused started return value: The information sent back, from a function started on a parallel component, is not reachable.
31. Infinite loops: Loops the code could not exit from.
32. Busy wait: Waiting for message in an event based system with polling.
33. Non-private private definitions: Public definitions used only internally.
34. Excessive rotation size: List rotation size should not exceed the size of the list.
35. Consequtive assignments to an entity: Assignments could be merged to a single assignment.
36. Sequential "if" statements: If possible should be changed to "if-else" conditions.
37. Size check in loop limit: The size of an unchanged list is checked in every iteration.
38. Reused loop variables: Loop variable declared and used outside the loop.
39. Unnecessary condition: The condition can be evaluated by the static analyzer.
40. Conditional complexity: Too large conditional logic blocks.
41. Explicit condition check: Explicitly check the value of a boolean condition.
42. Boolean evaluation with branching: All of the branches only set a single logical value.
43. Mergeable conditions: Consecutive conditionals do exactly the same operations.
44. If without else: In testing software all execution paths should be handled, at least logged.

45. Method with single condition: All statements of a function are in a single conditional.
46. Too many branches on a value: Switching on a value with consecutive conditionals.
47. Not written inout parameter: Reference passing used when not needed.
48. Not written out parameter: Result not calculated and passed back.
49. Not written variable: Variable declaration when constant would suffice.
50. Restrictable templates: Templates that could be more restricted based on their usage, but are not.
51. Dead code: Code fragment which is executed but not used anywhere.
52. Code commented out: Instead of removing it code was commented out.
53. Empty blocks: An empty code block.
54. Setverdict without reason: The testcase verdict is set without attached reason.
55. Variant outside Encodes: Encoding variants are specified without context.
56. Functions containing Stop: The execution is stopped inside a function, instead of the testcase.
57. Valueof used with value: The valueof function (used to convert a template to a value) is used with a value parameter.
58. Magic number: Numeric literals in the code.
59. Magic string: String literals inside the code.
60. XML tags in strings: XML encoding is simulated via string manipulation.
61. Nested block depth: The nesting of constructs exceeded a given level.
62. Indicent exposure: Too much of the module is exposed to the public.
63. Inappropriate intimacy: Dependencies on other module's implementation details. Functions using definitions only from an other module should be moved there. Members used only by a single external module should be moved there.
64. Feature envy: The function uses only an other module's attributes.
65. Divergent change: Changes touch completely different parts of a module.
66. Shotgun surgery: A change requires several changes in several modules.
67. PTC created, not started: A Parallel component is not started.
68. Isolated PTC: A parallel component is not connected to the test system.
69. Un-needed "runs on": There is no need for restricting a function to a specific component.

70. Contrieved complexity: Complex design patterns, where simpler would suffice.
71. Incorrect indentation: The code is not well indented.
72. Divergent naming of files: The names of files does not follow the naming conventions.
73. Incorrect pre-processability indication: Pre-processablity is not indicated in file extension.
74. Ordering of definitions: Definitions declared out of order.
75. Filling in values one-by-one: Structured value is filled in in several statements.
76. Private definitions published: A public function returns with a private definition creating a potential security hole.
77. Floating point equality check: Floating point numbers should not be compard directly.
78. Public/private keywords: The public/private keywords are used as identifiers.
79. Select without default branch: A select statement does not have "case else" branch.
80. Switch density: The ratio of branches are too high in the code.
81. Logic inversion: the whole conditional expression is negated.
82. Cyclometric complexity: The number of decision points in a method, plus one for the method entry.
83. NPath complexity: The number of acyclic execution paths in a method. Similar to Cyclometric complexity, but also takes into account the nesting of statements.
84. Break/continue usage: Break and continue statements are used incorrectly.
85. Unreachable code: A part of the code that can not be reached.
86. Using "*" for mandatory fields: Optionality is indicated for a mandatory field.

## B.2 Correlations among code smell data

| Code Smells | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 FIXME tags | 1.00 | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 2 TODO tags | 0.98 | 1.00 | | | | | | | | | | | | | | | | | | | | | | | | | |
| 3 Circular importation | 0.42 | 0.40 | 1.00 | | | | | | | | | | | | | | | | | | | | | | | | |
| 4 Empty statement block | 0.99 | 0.98 | 0.43 | 1.00 | | | | | | | | | | | | | | | | | | | | | | | |
| 5 If instead altguard | 0.99 | 0.97 | 0.43 | 0.98 | 1.00 | | | | | | | | | | | | | | | | | | | | | | |
| 6 If without else | 0.87 | 0.87 | 0.44 | 0.91 | 0.87 | 1.00 | | | | | | | | | | | | | | | | | | | | | |
| 7 Magic numbers | 0.98 | 0.96 | 0.47 | 0.99 | 0.96 | 0.90 | 1.00 | | | | | | | | | | | | | | | | | | | | |
| 8 Magic strings | 0.99 | 0.98 | 0.42 | 0.99 | 0.98 | 0.90 | 0.99 | 1.00 | | | | | | | | | | | | | | | | | | | |
| 9 Module name in definition | 0.86 | 0.85 | 0.39 | 0.90 | 0.86 | 0.99 | 0.89 | 0.90 | 1.00 | | | | | | | | | | | | | | | | | | |
| 10 Logic inversion | 0.97 | 0.97 | 0.43 | 0.99 | 0.95 | 0.92 | 0.98 | 0.99 | 0.93 | 1.00 | | | | | | | | | | | | | | | | | |
| 11 Definition should be private | 0.98 | 0.96 | 0.45 | 0.99 | 0.96 | 0.89 | 0.99 | 0.99 | 0.90 | 0.98 | 1.00 | | | | | | | | | | | | | | | | |
| 12 Readonly local variable | 0.68 | 0.69 | 0.35 | 0.72 | 0.67 | 0.67 | 0.72 | 0.68 | 0.66 | 0.74 | 0.67 | 1.00 | | | | | | | | | | | | | | | |
| 13 Readonly out formal parameter | -0.42 | -0.45 | -0.31 | -0.49 | -0.44 | -0.79 | -0.47 | -0.47 | -0.74 | -0.51 | -0.43 | -0.37 | 1.00 | | | | | | | | | | | | | | |
| 14 Readonly inout formal parameter | 0.97 | 0.97 | 0.46 | 0.98 | 0.96 | 0.86 | 0.98 | 0.97 | 0.85 | 0.97 | 0.97 | 0.75 | -0.42 | 1.00 | | | | | | | | | | | | | |
| 15 Size check in loop | 1.00 | 0.98 | 0.41 | 0.99 | 0.98 | 0.86 | 0.98 | 0.99 | 0.86 | 0.98 | 0.98 | 0.67 | -0.40 | 0.98 | 1.00 | | | | | | | | | | | | |
| 16 Switch on boolean | 0.98 | 0.97 | 0.39 | 0.98 | 0.95 | 0.81 | 0.97 | 0.97 | 0.81 | 0.97 | 0.97 | 0.68 | -0.33 | 0.97 | 0.99 | 1.00 | | | | | | | | | | | |
| 17 Too complex expression | 0.99 | 0.98 | 0.42 | 0.99 | 0.98 | 0.90 | 0.99 | 1.00 | 0.90 | 0.99 | 0.99 | 0.67 | -0.47 | 0.97 | 0.99 | 0.97 | 1.00 | | | | | | | | | | |
| 18 Too many parameters | 0.99 | 0.98 | 0.41 | 0.99 | 0.98 | 0.85 | 0.98 | 0.99 | 0.85 | 0.97 | 0.98 | 0.68 | -0.39 | 0.97 | 0.99 | 0.98 | 0.99 | 1.00 | | | | | | | | | |
| 19 Typename in definition | 0.94 | 0.93 | 0.42 | 0.93 | 0.95 | 0.80 | 0.92 | 0.95 | 0.80 | 0.92 | 0.96 | 0.56 | -0.32 | 0.93 | 0.96 | 0.93 | 0.95 | 0.93 | 1.00 | | | | | | | | |
| 20 Uncommented function | 0.97 | 0.95 | 0.47 | 0.98 | 0.96 | 0.95 | 0.98 | 0.98 | 0.95 | 0.98 | 0.98 | 0.68 | -0.57 | 0.95 | 0.97 | 0.94 | 0.98 | 0.96 | 0.92 | 1.00 | | | | | | | |
| 21 Uninitialized variable | 0.99 | 0.99 | 0.41 | 0.99 | 0.98 | 0.87 | 0.98 | 0.99 | 0.86 | 0.98 | 0.98 | 0.70 | -0.42 | 0.98 | 1.00 | 0.98 | 0.99 | 0.99 | 0.95 | 0.96 | 1.00 | | | | | | |
| 22 Unnecessary control | 0.86 | 0.87 | 0.44 | 0.91 | 0.88 | 1.00 | 0.89 | 0.90 | 0.98 | 0.92 | 0.88 | 0.67 | -0.80 | 0.86 | 0.86 | 0.82 | 0.90 | 0.85 | 0.80 | 0.94 | 0.87 | 1.00 | | | | | |
| 23 Unused function return values | 0.97 | 0.94 | 0.40 | 0.96 | 0.97 | 0.91 | 0.96 | 0.98 | 0.90 | 0.95 | 0.96 | 0.57 | -0.53 | 0.92 | 0.97 | 0.93 | 0.98 | 0.96 | 0.93 | 0.97 | 0.96 | 0.90 | 1.00 | | | | |
| 24 Unused global definition | 0.91 | 0.92 | 0.38 | 0.93 | 0.89 | 0.79 | 0.93 | 0.92 | 0.80 | 0.95 | 0.91 | 0.82 | -0.32 | 0.93 | 0.92 | 0.94 | 0.92 | 0.93 | 0.84 | 0.89 | 0.93 | 0.79 | 0.83 | 1.00 | | | |
| 25 Unused import | -0.72 | -0.72 | -0.43 | -0.75 | -0.75 | -0.87 | -0.74 | -0.75 | -0.84 | -0.73 | -0.74 | -0.34 | 0.79 | -0.70 | -0.72 | -0.64 | -0.76 | -0.70 | -0.73 | -0.81 | -0.71 | -0.87 | -0.84 | -0.49 | 1.00 | | |
| 26 Unused local definition | 0.04 | 0.05 | -0.11 | 0.01 | -0.01 | -0.32 | 0.02 | 0.00 | -0.28 | 0.02 | 0.01 | 0.34 | 0.69 | 0.09 | 0.05 | 0.14 | -0.01 | 0.09 | 0.01 | -0.11 | 0.07 | -0.32 | -0.17 | 0.31 | 0.61 | 1.00 | |
| 27 Visibility in definition | 0.98 | 0.97 | 0.38 | 0.97 | 0.95 | 0.83 | 0.97 | 0.98 | 0.83 | 0.96 | 0.97 | 0.64 | -0.36 | 0.96 | 0.99 | 0.98 | 0.98 | 0.99 | 0.94 | 0.95 | 0.98 | 0.82 | 0.94 | 0.93 | -0.67 | 0.10 | 1.00 |

TABLE B.1: The Pearson correlation values between the data series of the code smells. To save on space the numbers in the header represent the code smells, numbered in the first column.

# Appendix C

# Survey questions

## C.1 Mindset survey

Here are the mindset survey questions. The layout below is simplified to meet space limitations. We have noted within /* */ comments the different types of responses expected if not listed here.

### C.1.1 Generic information

1. Are you working for a multi-national company? (A company present in several countries.) /* yes-no */

2. How large is the company you are working for? (The number of employees working in your country.)

   (a) 1-10 employees (b) 11-50 employees (c) 51-150 employees (d) 151-500 employees (e) 501 - 1000 employees (f) 1000+ employees

3. How many people are you working with in your main project?

   (a) 1-3 (b) 4-7 (c) 8-14 (d) 15-30 (e) 30+

4. How long have you been working in your current position for?

   (a) 0-2 years (b) 3-5 years (c) 6-10 years (d) 10+ years

5. What is your predominant role or responsibility within your organization?

   (a) Development (b) Testing (c) Architect (d) Technical writing (e) Team leadership (f) Project management (g) Business operation/-support (h) Executive management (i) Managing of managers (j) Line management (k) Self-employed

6. What was your main task in the last year?

   (a) Requirement gathering (b) Research (c) System development (d) Writing conceptual information (e) Code editing (f) Code review (g) Deployment (h) Testing (i) Test review (j) Writing documentation (k) Maintenance (l) Managing the environment (m) Managing people (n) Administration (o) Managing Projects (p) Sales

7. What other responsibilities did you have beside your main task in the last year?

   (a) Requirement gathering (b) Research (c) System development (d) Writing conceptual information (e) Code editing (f) Code review (g) Deployment (h) Testing (i) Test review (j) Writing documentation

(k) Maintenance (l) Managing the environment (m) Managing people (n) Administration (o) Managing Projects (p) Sales

### C.1.2 Familiarity with different techniques

8. Which of the following software design patterns are you familiar with?

   (a) Builder (b) Factory (c) Singleton (d) Decorator (e) Composite (f) Proxy (g) Iterator (h) Chain of responsibility (i) State (j) Visitor (k) Strategy (l) Join (m) Lock (n) Message Design Pattern (o) Monitor (p) None of the above

9. Which of the following testing techniques are you familiar with?

   (a) Function testing (b) Boundary value anaysis (c) Decision table testing (d) Pairwise testing (e) Classification tree method (f) Statement testing (g) Branch testing (h) Exploratory testing (i) Fault attack with defect checklist (j) Error guessing (k) Cause-effect graph (l) Use-case testing (m) Path testing (n) Fault injection (o) Control flow analysis (p) Coding standard (q) Code metrics (r) Call graphs (s) Review (t) Walk-through (u) Inspection (v) None of the above

10. Which of the following techniques/methodologies are you familiar with?

    (a) Sequential development (b) Waterfall (c) V-model (d) Spiral model (e) Extreme programming (f) Scrum (g) Kanban (h) Agile (i) Test Driven Development (j) Feature Driven Development (k) Acceptance Test Driven Development (l) Continuous Integration (m) Integration Centric Engineering (n) Lean Development (o) 6 Sigma (p) Pair programming (q) CMMI (r) Planning poker (s) Refactoring (t) None of the above

11. Which of the following technical writing techniques are you familiar with?

    (a) Analysis of audience (b) Gathering specific vocabulary (c) Precise expressions (d) Clear design (e) Chain of new concepts (f) Review (g) i18n (h) L10n (i) Survey (j) User documentation (k) System documentation (l) Documentation Life Cycle (m) Problem-Method-Solution (n) Chronological structure (o) User testing (p) Camera-ready (q) S-V-O structure (r) Proofreading (s) Interview (t) Focus groups (u) None of the above

12. In your opinion how important is to have a a developer's mindset for your work? /* marks between 1 and 5 */

13. In your opinion how important is to have a tester's mindset for your work? /* marks between 1 and 5 */

14. In your opinion how important is to have a technical writer's mindset for your work? /* marks between 1 and 5 */

15. In your opinion how important is to have a management mindset for your work? /* marks between 1 and 5 */

### C.1.3 Gaining new knowledge

16. What are your main sources of gaining new knowledge?

    (a) Books (b) Research papers (c) Colleagues (d) Classes (e) Trainings (f) Vendor sites (g) Internet forums and blogs (h) Company intranet (i) Conferences (j) Other;

17. Which of the following resources did you use to learn last year?

    (a) Books (b) Research papers (c) Colleagues (d) Classes (e) Trainings (f) Vendor sites (g) Internet forums and blogs (h) Company intranet (i) Conferences (j) Other;

18. How much of the knowledge you need in your work have you acquired through formal training? (Percentage between 0 and 100)

19. How much of the knowledge you need in your work have you acquired through job training? (Percentage between 0 and 100)

20. How much of the knowledge you need in your work have you acquired through self-study? (Percentage between 0 and 100)

21. How much of the knowledge you need in your work have you acquired through trial and error? (Percentage between 0 and 100)

### C.1.4 Process and methodology related questions

22. In our company we are monitoring and evaluating the newest technologies/methodologies. /* marks between 1 and 5 */

23. When a new piece of technology/methodology is available we do extensive testing before introducing it into our processes. /* marks between 1 and 5 */

24. When a new activity/artifact is defined we establish sets of hypotheses that can be tested before work starts. /* marks between 1 and 5 */

25. When an activity is not done as specified we follow a defined process to improve. /* marks between 1 and 5 */

26. When we see a defective outcome despite all activity done as specified, we modify the processes. /* marks between 1 and 5 */

27. In my opinion in the last year I was idle ...% of my time: /* asking for percentage */

28. As far as I can tell, when someone is idle for long, our team is able to detect the situation and follow a defined process to modify or reassign activities. /* marks between 1 and 5 */

29. In my opinion in the last year I was overloaded ...% of my time: /* asking for percentage */

30. As far as I can tell, when someone is overloaded for long, our team is able to detect the situation and follow a defined process to modify or reassign activities. /* marks between 1 and 5 points */

31. If we find that a non-specific activity is needed, we learn why it is needed and redesign our processes. /* marks between 1 and 5 points */

32. In most cases in the processes we follow I find it clear what the next activity is . /* marks between 1 and 5 points */

33. I find it clear who has to carry out the next activity in the processes we follow. /* marks between 1 and 5 points */

34. When we plan to make a change we asses the current state of affair with scientific rigor. /* marks between 1 and 5 points */

35. When the result of a change is different from the expected we check how the change was done, what effects it had and redesign the change if needed. /* marks between 1 and 5 points */

### C.1.5 Anti-patterns

36. How familiar are you with design anti-patterns?

    (a) I have never heard of them (b) I have heard of them, but I'm not sure what they are (c) I know of them, but I'm not very concerned of them appearing in my work (d) I know and try to avoid them (e) I have a strong understanding and frequently use tools to detect and remove anti-patterns

37. How concerned are you about the presence of anti-patterns in your products? /* marks between 1 and 5 points */

38. How often do you work on existing products to improve their internal quality without changing their external behaviour?

    (a) Never (b) Seldom (c) Sometimes (d) When absolutely necessary (e) On a regular basis (f) Such work is planned and done a formal activity.

39. Is working on existing products to improve their internal quality supported by your organization? (Only internal quality, without changing external behaviour)

    (a) No (b) In theory (c) Tools are available (d) When we have free time (e) We have allocated time for this kind of work in our processes

40. If internal quality improvement is done, when is it done?

    (a) We don't perform internal quality improvements (b) When there are issues, we correct it (c) When we notice a possibility to improve we take it immediately (d) We have pre-planned sessions and work lists for internal quality improvements

41. In the last year how many times did you have to produce solutions you felt they were of low quality in order to satisfy short term needs?

    (a) Never (b) 1-2 times (c) 3-5 times (d) 6-10 times (e) 10+ times

### C.1.6 Static analysis and traceability

42. Which tool supported static analysis techniques are used in your organization?

    (a) Checking of static metrics (b) Checking of coding standards (c) Control flow analysis (d) Data flow analysis (e) Other tools supporting static analysis (f) Our techniques are not tool supported

43. Do you have manual code reviews for internally developed products? /* yes - no question */

44. How long does a manual review take? (In minutes): /* expecting a number */

45. In your opinion, which stage could be the issues found in the last year traced back to?

    (a) Requirement collection (b) Concept/System design (c) Code writing (d) Documentation (e) Review (f) User support (g) Management of tasks (h) Management of people

46. How do you judge if a specification is out-of-date? /* free text expected */

47. What kind of direct contact do you have with your users?

    (a) Phone contact (b) Chat application (Skype, Messenger, etc.) (c) Direct Email (d) Formal meetings held periodically (e) We have no direct contact to users (f) Other: /* free text expected */

## C.2 Titanium survey

- Which of the DAG/reverse DAG, ISOM, Kamada-Kawai, Frushterman-Reingold layouts do you find most useful in your daily work?
- Are nodes, extracted to the 0th level, simple to notice?
- Is the Dag displaying of not-imported modules in the first row, really helping to find unnecessary modules?
- Is the visualization of circles easy to notice?
- Which of the grouping or graph generating clustering is more useful for you?
- How useful do you find the folder based clustering?
- How useful do you find the name based clustering?
- Is it important for you, that these tools are integrated into the development environment?
- How intuitive was the usage of main and satellite views?
- How much effort was need to learn the views?
- How useful do you find module dependency visualization?

[1]**ADATLAP**
**a doktori értekezés nyilvánosságra hozatalához**

**I. A doktori értekezés adatai**
A szerző neve:..... Szabados Kristóf .....
MTMT-azonosító:..... 10040393.....
A doktori értekezés címe és alcíme: Quality Aspects of TTCN-3 Based Test Systems
DOI-azonosító[2]: 10.15476/ELTE.2017.159
A doktori iskola neve:..... Informatika Doktori Iskola .....
A doktori iskolán belüli doktori program neve:... Információs Rendszerek Program ...
A témavezető neve és tudományos fokozata:... Dr. Habil. Kovács Attila .....
A témavezető munkahelye:..... ELTE, Komputer Algebra Tanszék .....

**II. Nyilatkozatok**
**1.** A doktori értekezés szerzőjeként[3]

a) hozzájárulok, hogy a doktori fokozat megszerzését követően a doktori értekezésem és a tézisek nyilvánosságra kerüljenek az ELTE Digitális Intézményi Tudástárban. Felhatalmazom az Informatika Doktori Iskola hivatalának ügyintézőjét, Boda Annamáriát, hogy az értekezést és a téziseket feltöltse az ELTE Digitális Intézményi Tudástárba, és ennek során kitöltse a feltöltéshez szükséges nyilatkozatokat.

b) kérem, hogy a mellékelt kérelemben részletezett szabadalmi, illetőleg oltalmi bejelentés közzétételéig a doktori értekezést ne bocsássák nyilvánosságra az Egyetemi Könyvtárban és az ELTE Digitális Intézményi Tudástárban;[4]

c) kérem, hogy a nemzetbiztonsági okból minősített adatot tartalmazó doktori értekezést a minősítés (*dátum*)-ig tartó időtartama alatt ne bocsássák nyilvánosságra az Egyetemi Könyvtárban és az ELTE Digitális Intézményi Tudástárban;[5]

d) kérem, hogy a mű kiadására vonatkozó mellékelt kiadó szerződésre tekintettel a doktori értekezést a könyv megjelenéséig ne bocsássák nyilvánosságra az Egyetemi Könyvtárban, és az ELTE Digitális Intézményi Tudástárban csak a könyv bibliográfiai adatait tegyék közzé. Ha a könyv a fokozatszerzést követőn egy évig nem jelenik meg, hozzájárulok, hogy a doktori értekezésem és a tézisek nyilvánosságra kerüljenek az Egyetemi Könyvtárban és az ELTE Digitális Intézményi Tudástárban.[6]

**2.** A doktori értekezés szerzőjeként kijelentem, hogy

a) az ELTE Digitális Intézményi Tudástárba feltöltendő doktori értekezés és a tézisek saját eredeti, önálló szellemi munkám és legjobb tudomásom szerint nem sértem vele senki szerzői jogait;

b) a doktori értekezés és a tézisek nyomtatott változatai és az elektronikus adathordozón benyújtott tartalmak (szöveg és ábrák) mindenben megegyeznek.

**3.** A doktori értekezés szerzőjeként hozzájárulok a doktori értekezés és a tézisek szövegének plágiumkereső adatbázisba helyezéséhez és plágiumellenőrző vizsgálatok lefuttatásához.

Kelt: BUDAPEST, 2017. 10. 25

Szabados Kristóf
**a doktori értekezés szerzőjének aláírása**

---

[1] Beiktatta az Egyetemi Doktori Szabályzat módosításáról szóló CXXXIX/2014. (VI. 30.) Szen. sz. határozat. Hatályos: 2014. VII.1. napjától.
[2] A kari hivatal ügyintézője tölti ki.
[3] A megfelelő szöveg aláhúzandó.
[4] A doktori értekezés benyújtásával egyidejűleg be kell adni a tudományági doktori tanácshoz a szabadalmi, illetőleg oltalmi bejelentést tanúsító okiratot és a nyilvánosságra hozatal elhalasztása iránti kérelmet.
[5] A doktori értekezés benyújtásával egyidejűleg be kell nyújtani a minősített adatra vonatkozó közokiratot.
[6] A doktori értekezés benyújtásával egyidejűleg be kell nyújtani a mű kiadásáról szóló kiadói szerződést.