DE GRUYTER OPEN Acta Univ. Sapientiae, Informatica **8,** 2 (2016) 216–240

DOI: 10.1515/ausi-2016-0010

Internal quality evolution of a large test system–an industrial study

Attila KOVÁCS
Eötvös Loránd University
email: attila.kovacs@inf.elte.hu

Kristóf SZABADOS[1]
Eötvös Loránd University
email:
kristof.szabados@ericsson.com

Abstract

This paper presents our empirical observations related to the evolution of a large automated test system. The system observed is used in the industry as a test tool for complex telecommunication systems, itself consisting of more than one million lines of source code. This study evaluates how different changes during the development have changed the number of observed Code Smells in the test system. We have monitored the development of the test scripts and measured the code quality characteristics over a five years period.

The observations show that the introduction of continuous integration, the existence of tool support for quality improvements in itself, changing the development methodologies (from waterfall to agile), changing technical and line management structure and personnel caused no measurable change in the trends of the observed Code Smells. Internal quality improvements were achieved mainly by individuals intrinsic motivation. Our measurements show similarities with earlier results on software systems evolutions presented by Lehman.

Computing Classification System 1998: D.2.2, D.2.9
Mathematics Subject Classification 2010: 68N99
Key words and phrases: code smells; empirical study; software evolution; test systems; Lehman's laws; TTCN-3

[1]Corresponding author

1 Introduction

Do we really know how to build large test sets? Do we really know how test systems evolve, how can their development be managed, their quality ensured?

Nowadays the usage of software systems belongs to the everyday life of the society, yet testing these systems is still a challenging and not really understood activity. Software helps in navigating to locations, supports communication with other people, drives the production, distribution and consumption of energy resources. Software controls companies, trades on the markets, takes care of people's health.

To support the testing these systems need, ETSI[2] developed the TTCN-3[3] language, which can be used in testing of reactive systems. By now test systems developed by ETSI, 3GPP[4] and in the industry, have evolved to be comparable to the tested systems in both size and complexity ([26]). Standardized test systems were shown to contain design problems similar to those present in other (C / C++ / Java) systems ([29, 28]).

In this paper we show empirical observations on the evolution of two large test systems developed in the industry and our measurements on the number of code smells in them. We ask the following research questions: Was the number of measured code smells affected by the introduction of Continuous Integration (RQ1), by tool support for detecting code smells (RQ2), by the merging of 2 test systems (RQ3), by doing the development using different methodologies (RQ4), by changing leaders on the project (RQ5) ? Our final research question is: Do code smells in test systems follow predictable patterns during the system's evolution (RQ6) ?

In this study we examine the evolution of TTCN-3 test systems from a software quality point of view. In our research we treat test systems as software products used for testing, rather than tests. We present historical information on changes in line and project management, development practices, organizational and technical structures, tool support that happened during their five years development period. By comparing our measurements with historical information we show how the evolution could affect the quality of the observed large scale test system.

[2]European Telecommunication Standardization Institute
[3]Testing and Test Control Notation Version 3
[4]3rd Generation Partnership Project

1.1 Structure of this paper

This paper is organized as follows. In Section 2 we present earlier results related to this subject. Section 3 contains the history of the studied projects and the measurement environment. In Section 4 we analyze the measured data and correlate the measurements to publicly known items. Section 5 presents the measured results from the enterprise' point of view (significance of development methods, leadership styles, tool support) to emphasize their influences. Section 6 lists the factors that might be a threat to the validity of our results. Finally, Section 7 summarizes our findings.

2 Previous work

Before presenting our findings it is necessary to understand the importance and limitations of code smells, software evolution and the state of how this knowledge is translated to testing.

2.1 Code smell studies

Code smells were introduced by Fowler [4] as issues that are not necessarily technically incorrect codes and do not disable the program from functioning, but might indicate architectural problems or misunderstandings, issues which are very hard to detect. Since then, the initial list of 22 code smells has been extensively extended (see e.g. [33, 19, 21]), 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 on code smells revealed that smelly codes in software systems are changed more frequently than other codes ([9, 22]). Moser at al. found [20] 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.

On the other hand the value of code smells has been questioned by many. Yamashita et al. found [37] that only 30% of the maintenance problems were related to files containing code smells. Sjøberg at al. found [25] that none of the code smells they investigated was significantly associated with increased maintenance effort when adjusted by file size. Macia et al. observed [18] that more than 60% of the automatically detected code anomalies were not correlated with architectural problems.

In order to understand software aging better the lifespan of code smells was

studied by many (see e.g. [23]). Chatzigeorgiou et al. published [1] 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 concluded [24] that developers might be aware of code smells, but are usually not concerned by their presence. In each system inspected there were only one or two developers who resolved code smell instances intentionally, or resolved significantly more instances than others (possibly unintentionally).

In their 2013 paper Yamashita et al. [35] 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 did not know about code smells, nor did they care. Those who were at least somewhat concerned about code smells indicated difficulties with obtaining organizational support and tooling. In their empirical studies ([34, 36]) 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.

2.2 Software evolution studies

Lehman [14] 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* the following are the most relevant for this study [16]:

- Law 2: "As an E-type[5] 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 [15], and Lawrence [10] found that commercial systems have a clear linear growth, viewed over a number of releases. Izurieta and

[5]systems actively used and embedded in a real world domain.

Bieman found [6] that Open Source Software products FreeBSD and Linux also appear to grow at similar rates.

Turski showed ([32]) that the gross growth trends can be predicted, with a mean absolute error of order 6%, with

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

where S_i is the system size at the i-th measurement, and $\hat{e}$ can be calculated as $(S_{i-1} - S_1)/(\sum_{k=1}^{i-1} 1/S_k^2)$.

There are plenty of researches ([17, 13, 8, 7, 5]) in which the authors show that the laws seem to be supported by solid evidence. But applying them currently requires the understanding of human, technical, usage and organizational contexts of the measures, they were derived from.

2.3 Test quality and evolution studies

Deursen at al. [3] noticed while working on a Java project 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.

Zaidman et al. [38] witnessed both phased and synchronous co-evolution of tests and production codes.

Zeiss et al. [39] published a model for TTCN-3 test specification derived from ISO 9126, by translating the quality standard for testing.

2.4 Our contributions

In our long term research we explore similarities between systems of tests and software systems. We look at tests as software systems, re-interpreting test systems and script as software products.

In [26] we have shown that automated test systems written in TTCN-3 can grow large and complex similar to the structures studied in [31]. In [29] we have defined 86 code smells for TTCN-3 and their relations to international software quality standards. In order to understand the quality of such huge test systems 35 selected code smells were implemented and measured on 16 projects.

The updated list of code smells, used in this measurement, can be found at [30].

To the best of our knowledge the evolution of code smells in test systems was not yet studied in the domain of testing communication systems. We have also not found any work presenting the relation between real world events and what effects they had on the quality of test suites.

3 History of the studied systems

3.1 Background

Current test systems have grown large with 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 (ex. the software layer converting between abstract TTCN-3 messages and actual bit stream messages). For this reason in this article 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[6] products. At the end of 2012 these two test systems were merged into one, which we will call the `Merged` System.

`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).

Titanium is our open source ([27]), static code analyzer, developed as part of our research to support detecting issues in TTCN-3 source codes.

3.2 History of the tracked systems

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

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

- 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 for development, led by an experienced Scrum Master. Strong coordination appeared 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 readonly variables, inout and out parameters, unused local definitions

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

3.3 Subjective information

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` have 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.

3.4 Trainings on Code Smells and usage of Titanium

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.

3.5 Effort

Table 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[7] to represent the starting and closing of the year.

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 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 almost 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`.

4 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 at [30]) present.

4.1 Size

We analyzed the size growth of `System-1` and `Merged` systems measured in LOC. Figure 1 shows the measured data[8] and a quadratic trend line fitted.

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

[8] Measuring the lines of code was an afterthought in our case. For System-1 we measured the lines of code of released software versions, for Merged we show monthly measurement

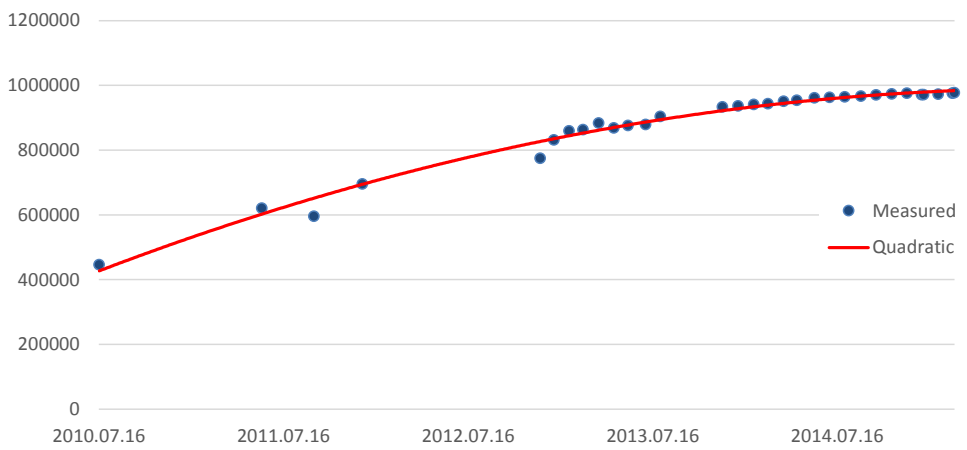

Figure 1: Size evolution of the `System-1` and `Merged` systems.

When we used the Lehman's prediction according to equation (1) on the lines of code in `Merged`, we measured a maximal absolute error between the measured data and the predicted model is about 3%.

4.2 Correlations among code smells

For each possible pair of code smells we calculated the Pearson correlation between the data series of the code smells ([30]) on the `Common` + `System-1` + `Merged` system evolution (Table 2). We excluded code smells having less than 50 occurrences at every measurement point during the development of the systems, as even small changes can appear to break trends using such small numbers. Based on the correlation values the code smells could be separated into 3 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, readonly 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.*

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 2: The Pearson correlation 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.

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, readonly 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.*

4.3 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.

4.3.1 First correlation group

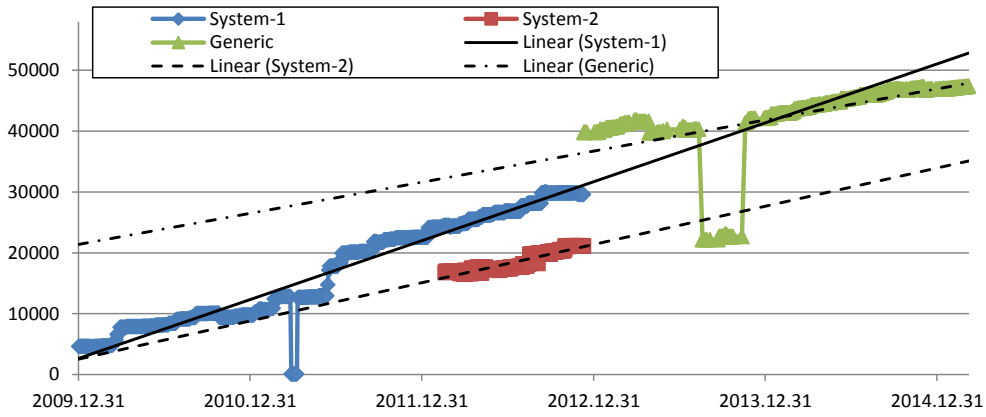

Figure 2: *Number of magic string* issues and its linear approximations.

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 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. Directly 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` 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.

4.3.2 Second correlation group

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

In case of the *Module name in definition* code smell (Figure 3) the trends of `System-1` and `System-2` seems to be added together, and following the growth trend of `System-2`. After the merge the smell occurences of `Merged` followed the growth of `System-2`.

In case of the *Readonly local variable* code smell (Figure 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.

The *Typename in definition* trends (Figrure 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`

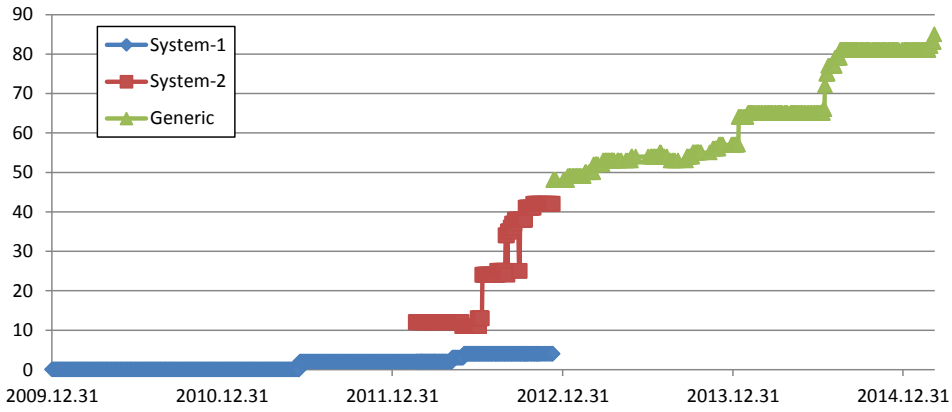

Figure 3: *Module name in definition* smell trends

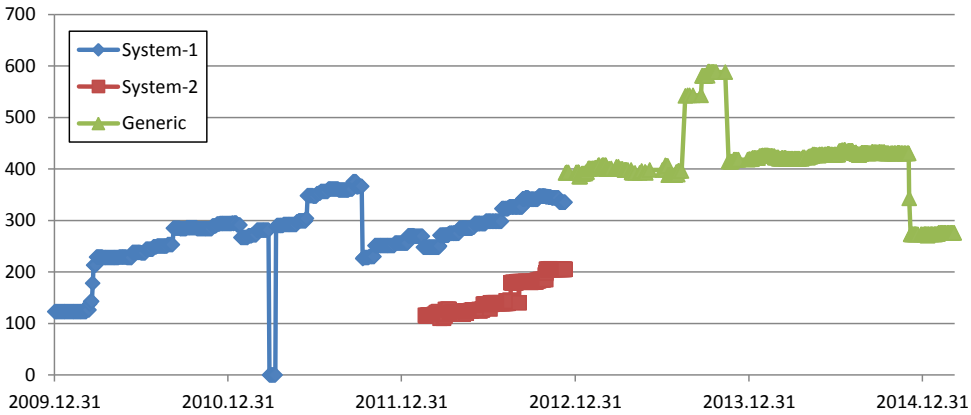

Figure 4: *Readonly local variable* smell trends

continued in `Merged` (Figure 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*.

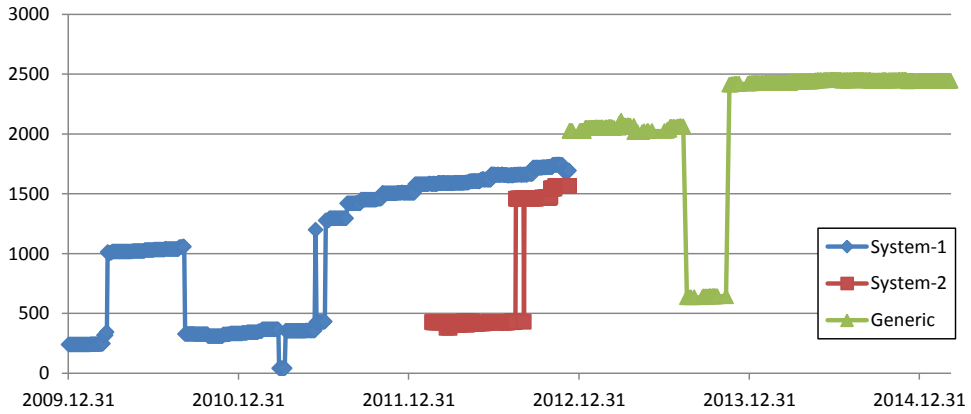

Figure 5: *Typename in definition* smell trends

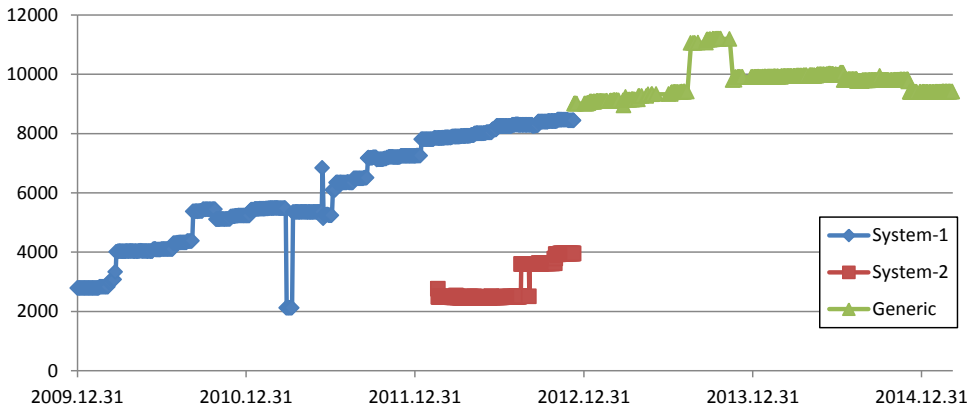

Figure 6: *Unused global definition* smell trends

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 is reported for every module in the 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, in `System-2` chaotic till

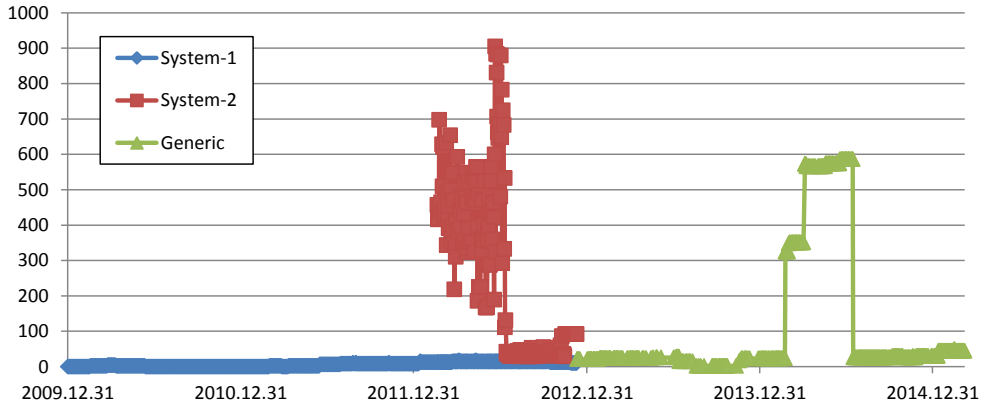

Figure 7: *Circular importation* smell trends

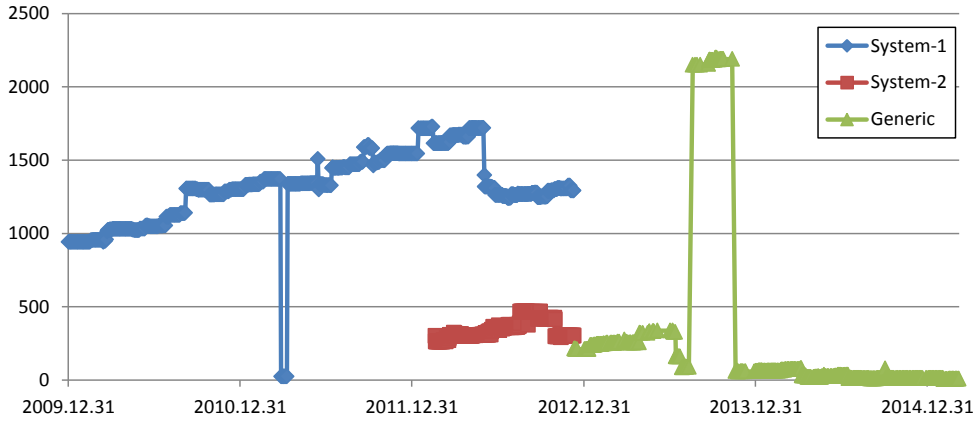

Figure 8: Number of *unused imports* smell trends.

the half of that year. Which continued with 2 in 2013 and 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. Only to be reduced to 45 during the "Green Day" event.

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

4.3.3 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 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 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 Analysis of our research questions

5.1 Was the number of measured code smells affected by the introduction of Continuous Integration (RQ1)?

Continuous Integration was introduced together with the Agile methodology. As many systems had to adapt to it the process took months, with fine tuning happening even at the time of writing this article. Quality checking 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 checking 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 ([1, 24, 35]).

5.2 Was the number of measured code smells affected by the introduction of tool support for detecting Code Smells (RQ2) ?

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 analyzed 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 ([24]).

5.3 Was the number of measured code smells affected by the merging of 2 test systems (RQ3) ?

We measured that the merge increased the amount of code smells present and also decreased their previous growth rate.

These results align with the 5th law of software evolution ([16]) and other earlier results ([1, 24, 35]).

It is interesting to note, that the growth of the merged system is between the original growths of the two systems it consists of. At the time of writing, we do not know whether this growth rate will stay longer or will follow one of the original system's growth rate.

5.4 Was the number of measured code smells affected by the different development methodologies (RQ4) ?

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.

5.5 Was the number of measured code smells affected by changing leaders of the projects (RQ5) ?

Conway's law [2] 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.

5.6 Do code smells in test systems follow predictable patterns during the system's evolution (RQ6) ?

In this section we show how our findings detailed in section 4 relate to Lehman's laws of Software Evolution ([16]).

- 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 [12] 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 [17] 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.

6 Threats to validity

This study might suffer from the usual threats to external validity. There might be limits to generalizing our results beyond our settings (programming language used, project setups and possible industry specific effects).

This study was performed on two test systems, developed at the same organization. The field of software evolution studies has limited information sources. Publications in the field analyze only a few open source systems ([6, 10]) and few commercial systems ([11, 17]). Our efforts are an addition to the growing body of knowledge to this field.

To the best of our knowledge these are the first results for the evolution of test systems from software quality point of view, and also the first observation of the effects of products merging. Although it is a valid question if our results can be generalized to other testing languages and domains of software development, we believe this to be true as our results align with previous results in the field of software evolution ([5, 6, 7, 8, 10, 11, 13, 15, 16, 17, 32]).

The study might suffer from not measuring the metrics which were changed by the historical happenings. We have measured several code smells and presented our observations of their changes in this article. These code smells were either collected from a wide range of tools supporting other languages and

adapted to TTCN-3, or defined by us based on our earlier observations related to the language ([29]). We believe that these metrics are correctly measuring internal quality and exhaustive for the TTCN-3 language.

This study also faces the threat of delayed influence: as the work on the studied systems is still going on, it could happen that the influence of some change in the past, will only appear after the publication of this paper. We don't believe this to be a big threat, as the projects studied have been in development for 5 years, our tool support appeared 3 years ago and we have organized several code improvement special events in the last 2 years.

It is an unlikely but theoretically possible scenario that all changes happened at the right time: the changes were necessary to keep the rate of growth; all transitions were smooth and all changes stack up to keep up the same rate of growth.

7 Summary

We have previously defined ([29]) several code smells for test systems written in TTCN-3 and have shown ([28]) that publicly available TTCN-3 test systems have room for improvement. We have also already shown ([26]), that test systems written in TTCN-3, can become large and complex structures. In this article we studied the long term evolution of a large test system in the industry.

We have monitored the development of a test system and measured the code quality characteristics for a five years period at our industry partner. 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. We can conclude that the development of the observed test system follows predictable tendencies.

Just like Lehman's law predicted and observed in [1, 35].

The presence of tool support only made a difference when code smell reductions were the target of personal motivations. According to our observations the best way to improve a software's internal quality is to provide people with dedicated time and tools. This way people, who were already motivated [34, 24, 36, 35], could focus on a few lines of the source code instead of analyzing all of it by hand. This phenomenon was also observed in [24].

Our observation on the evolution of the studied test systems show similarity with the evolution of software systems. This is the main conclusion of the paper.

Acknowledgements

We thank the referee for providing constructive comments and help in improving the contents of this paper.

The authors would like to thank the Faculty of Informatics of Eötvös Loránd University for supporting this research.

We would also like to thank Gábor Jenei, Dániel Poroszkai and Dániel Góbor for their help in implementing features that were crucial to our investigation. Their work allowed us to quickly process large amount of data.

References

[1] A. Chatzigeorgiou, A. Manakos, Investigating the evolution of bad smells in object-oriented code, *Proc. 2010 Seventh International Conference on the Quality of Information and Communications Technology*, QUATIC'10, pp. 106–115, Washington, DC, USA, 2010. IEEE Computer Society. ⇒219, 232, 233, 236

[2] M. E. Conway, How do committees invent?, *Datamation*, **14** ,5 (1968) 28–31. http://www.melconway.com/research/committees.html [accessed 26-Aug-2015]. ⇒234

[3] A. v. Deursen, L. Moonen, A. v. d. Bergh, G. Kok, Refactoring test code, *Proc. 2nd International Conference on Extreme Programming and Flexible Processes* (XP2001), pp. 92–95. University of Cagliari, 2001. ⇒220

[4] M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ⇒218

[5] A. Israeli, D. G. Feitelson, The linux kernel as a case study in software evolution, J. Syst. Softw., **83,** 3 (2010) 485–501. ⇒220, 235

[6] C. Izurieta, J. Bieman, The evolution of freebsd and linux, *Proc. 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, ISESE'06, pp. 204–211, New York, NY, USA, 2006. ACM. ⇒220, 235

[7] K. Johari, A. Kaur, Effect of software evolution on software metrics: An open source case study, SIGSOFT Softw. Eng. Notes, **36,** 5 (2011) 1–8. ⇒220, 235

[8] C. F. Kemerer, S. Slaughter, An empirical approach to studying software evolution, IEEE Trans. Softw. Eng., **25,** 4, (1999) 493–509. ⇒220, 235

[9] F. Khomh, M. Di Penta, Y.-G. Gueheneuc, An exploratory study of the impact of code smells on software change-proneness, *Proc. 16th Working Conference on Reverse Engineering*, WCRE'09, pp. 75–84, Washington, DC, USA, 2009. IEEE Computer Society. ⇒218

[10] M. J. Lawrence, An examination of evolution dynamics, *Proc. 6th International Conference on Software Engineering*, ICSE'82, pp. 188–196, Los Alamitos, CA, USA, 1982. IEEE Computer Society Press. ⇒219, 235

[11] M. M. Lehman, *The programming process*, 1969. IBM Research Report RC 2722. ⇒235

[12] M. M. Lehman, Laws of software evolution revisited, *Proc. 5th European Workshop on Software Process Technology*, EWSPT '96, pp. 108–124, London, UK, UK, 1996, Springer-Verlag. ⇒235

[13] M. M. Lehman, *Feast/2 final report – grant number gr/m44101*, 2001. ⇒220, 235

[14] M. M. Lehman, J. F. Ramil, Towards a theory of software evolution - and its practical impact (working paper), *Proc. Intl. Symposium on Principles of Softw. Evolution (invited talk)*, ISPSE 2000, 1-2 Nov, pp. 2–11. Press, 2000. ⇒219

[15] M. M. Lehman, J. F. Ramil, Evolution in software and related areas, *Proc. 4th International Workshop on Principles of Software Evolution*, IWPSE'01, pages 1–16, New York, NY, USA, 2001, ACM. ⇒219, 235

[16] M. M. Lehman, J. F. Ramil, Rules and tools for software evolution planning and management, *Ann. Softw. Eng.,* **11,** 1 (2001) 15–44. ⇒219, 233, 234, 235

[17] M. M. Lehman, J. F. Ramil, D. E. Perry, On evidence supporting the feast hypothesis and the laws of software evolution, *Proc. 5th International Symposium on Software Metrics*, METRICS '98, pp. 84 –, Washington, DC, USA, 1998. IEEE Computer Society. ⇒220, 235

[18] I. Macia, J. Garcia, D. Popescu, A. Garcia, N. Medvidovic, A. von Staa, Are automatically-detected code anomalies relevant to architectural modularity?: An exploratory analysis of evolving systems, *Proc. 11th Annual International Conference on Aspect-oriented Software Development*, AOSD '12, pp. 167–178, New York, NY, USA, 2012, ACM. ⇒218

[19] N. Moha, Y.-G. Gueheneuc, L. Duchien, A.-F. Le Meur, Decor: A method for the specification and detection of code and design smells, IEEE Trans. Softw. Eng., **36,** 1 (2010) 20–36. ⇒218

[20] R. Moser, P. Abrahamsson, W. Pedrycz, A. Sillitti, G. 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*, pp. 252–266, Springer-Verlag, Berlin, Heidelberg, 2008. ⇒218

[21] H. Neukirchen, M. Bisanz, Utilising code smells to detect quality problems in ttcn-3 test suites, *Proc. 19th IFIP TC6/WG6.1 International Conference, and 7th International Conference on Testing of Software and Communicating Systems*, TestCom'07/FATES'07, pp. 228–243, Berlin, Heidelberg, 2007, Springer-Verlag. ⇒218

[22] S. Olbrich D. S. Cruzes, V. Basili, N. Zazworka, The evolution and impact of code smells: A case study of two open source systems, *Proc. 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ESEM '09, pp. 390–400, Washington, DC, USA, 2009. IEEE Computer Society. ⇒218

[23] D. L. Parnas, Software aging, *Proc. 16th International Conference on Software Engineering*, ICSE '94, pp. 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press. ⇒219

[24] R. Peters, A. Zaidman, Evaluating the lifespan of code smells using software repository mining, *Proc. 2012 16th European Conference on Software Maintenance and Reengineering*, CSMR'12, pp. 411–416, Washington, DC, USA, 2012. IEEE Computer Society. ⇒219, 232, 233, 236

[25] D. I. K. Sjoberg, A. Yamashita, B. Anda, A. Mockus, T. Dyba, Quantifying the effect of code smells on maintenance effort, *IEEE Trans. Softw. Eng.*, **39,** 8 (2013) 1144–1156. ⇒218

[26] K. Szabados, Structural analysis of large ttcn-3 projects. *Proc. 21st IFIP WG 6.1 International Conference on Testing of Software and Communication Systems and 9th International FATES Workshop*, TESTCOM '09/FATES '09, pp. 241–246, Berlin, Heidelberg, 2009, Springer-Verlag. ⇒217, 220, 236

[27] K. Szabados, *Titanium*, https://projects.eclipse.org/proposals/titan, 2015. [Online; accessed 26-Aug-2015]. ⇒221

[28] K. Szabados, A. Kovács, Advanced ttcn-3 test suite validation with titan, *Proc. 9th Conference on Applied Informatics*, pp. 273–281, 2014. ⇒217, 236

[29] K. Szabados, A. Kovács, Test software quality issues and connections to international standards, *Acta Universitatis Sapientiae, Informatica*, **5,** 1 (2014) 77–102. ⇒217, 220, 236

[30] K. Szabados and A. Kovács, *Up-to-date list of code smells*, `http://compalg.inf.elte.hu/~attila/TestingAtScale.htm`, 2015. [Online; accessed 26-Aug-2015]. ⇒220, 224, 225

[31] C. Taube-Schock, R. J. Walker, I. H. Witten, Can we avoid high coupling?, *Proc. 25th European Conference on Objectoriented Programming*, ECOOP'11, pp. 204–228, Berlin, Heidelberg, 2011, Springer-Verlag. ⇒220

[32] W. M. Turski, The reference model for smooth growth of software systems revisited, *IEEE Trans. Softw. Eng.,* **28,** 8, (2002) 814–815. ⇒220, 235

[33] E. Van Emden, L. Moonen, Java quality assurance by detecting code smells, *Proc. Ninth Working Conference on Reverse Engineering* (WCRE'02) pp. 97–106, Washington, DC, USA, 2002. IEEE Computer Society. ⇒218

[34] A. Yamashita, L. Moonen, Do code smells reflect important maintainability aspects?, *Proc. 2012 IEEE International Conference on Software Maintenance*, ICSM '12, pp. 306–315,Washington, DC, USA, 2012. IEEE Computer Society. ⇒219, 236

[35] A. Yamashita, L. Moonen, Do developers care about code smells? an exploratory survey, *Proc. 20th Working Conference on Conference: Reverse Engineering*, pp. 242–251. IEEE Computer Society, 2013. ⇒219, 232, 233, 236

[36] A. Yamashita, L. Moonen, Exploring the impact of inter-smell relations on software maintainability: An empirical study, *Proc. 2013 International Conference on Software Engineering*, ICSE '13, pp. 682–691, Piscataway, NJ, USA, 2013. IEEE Computer Society Press. ⇒219, 236

[37] A. Yamashita, L. Moonen, *To what extent can maintenance problems be predicted by code smell detection? - an empirical study*, *Inf. Softw. Technol.*, **55,** 12 (2013) 2223–2242. ⇒218

[38] A. Zaidman, B. Rompaey, A. Deursen, S. Demeyer, Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining, *Empirical Softw. Engg.*, **16,** 3 (2011) 325–364. ⇒ 220

[39] B. Zeiß, D. Vega, I. Schieferdecker, H. Neukirchen, J. Grabowski, Applying the ISO 9126 Quality Model to Test Specifications–Exemplified for TTCN-3 Test Specifications, *Software Engineering 2007*, *Lecture Notes in Informatics*, Copyright Gesellschaft für Informatik, Mar. 2007. ⇒ 220

Received: June 9, 2016 • Revised: August 20, 2016