Technical Debt of Standardized Test Software

Kristóf Szabados
Eötvös Loránd University, Budapest, Hungary
Kristof.Szabados@ericsson.com

Attila Kovács
Eötvös Loránd University, Budapest, Hungary
Attila.Kovacs@inf.elte.hu

***Abstract*—Recently, technical debt investigations became more and more important in the software development industry. In this paper we show that the same challenges are valid for the automated test systems. We present an internal quality analysis of standardized test software developed by ETSI and 3GPP, performed on the systems publicly available at `www.ttcn-3.org`.**

I. Introduction

As the size and complexity of software systems grow, so do their test systems. It is known that test architectures in the telecom area are comparable to the tested systems in both size and complexity [1]. Yet, these test systems received less attention in the past.

In a previous study [2] standardized test suites – available at `www.ttcn-3.org` – were analyzed. We found significant syntactic and semantic problems. We also found a large number of possible internal quality problems. In this article we provide estimations on the cost of fixing the internal quality issues found in the previously examined test software.

This paper is organized as follows. In Section II we present earlier works related to this subject. Section III-A presents our method for collecting estimates and the collected data. Section III-B shows our technical debt values for the measured projects. Section III-C deals with the validity of our results. Finally, Section IV summarizes our findings and Section V offers ideas for further research.

II. Definitions and previous work

Technical Debt: The term *technical debt* was first used by Cunningham [3] 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...*".

Recently, technical debt became a major concern. Griffith et al. conducted a study [4] showing that different forms of technical debt can have significant to strong correlation with reusability, understandability, effectiveness and functionality. Holvitie et al. found [5] that in the industry almost 9 out of 10 technical debt instances reside in the implementation and that agile practices close to the implementation are felt by practitioners to reduce or manage technical debt by their respondents. Ho et al. proposed an approach [6] which could help product managers to decide the release date of a product.

Ramasubbu et al. investigated [7] 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 results in poor customer satisfaction in the short term, but pays off on the long term with significantly higher software quality and customer satisfaction.

The study of technical debt related articles of Li at el. [8] shows that although the term "technical debt" became widespread, different people use it in different ways, leading to ambiguous interpretations. They also pointed out the need for more empirical studies on technical debt and how to apply specific approaches in industrial settings.

Code Smells: In this article we use *code smells* to measure the technical debt of software systems. *Code smells* were introduced by Fowler [9] as issues in the source code that might indicate architectural problems or misunderstandings, issues which are very hard to detect any other way.

Empirical work on code smells showed [10][11] that smelly codes in software systems were changed more frequently than other classes.

Moser at al. found [12] 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.

Code Smell Debates: Yamashita et al. [13] found in their survey that 32% of the respondents did not know about code smells nor did they care. Respondents at least somewhat concerned about code smells indicated difficulties with obtaining organizational support and tooling. They also observed [14] that code smells covered only some of the maintainability aspects considered important by developers. Developers did not take any conscious action to correct bad smells that were found in the code.

Quality Debt of Test Systems: It was shown in [1] that the size of automated test systems written in TTCN-3[1] may increase yielding large and complex systems. In [15] it was exposed how ISO/IEC 9126 and ISO/IEC 25010 software quality models can be applied to software systems that are used for testing. 86 code smells were defined, analyzed and categorized according to the quality models. 35 selected code smells were implemented and measured on 16 projects in order to understand the quality of such systems. Some of them were company internal, while others were created by standardization bodies.

In [2] the software quality of test suites available at `www.ttcn-3.org` were analyzed. It was shown that even the public testsuites created by ETSI[2] – the same organization

[1]Testing and Test Control Notation Version 3
[2]European Telecommunication Standardization Institute

behind the TTCN-3 language – contain a large number of code smell instances.

III. Technical Debt Analysis

A. Estimation

After exploring the test system quality issues in [2] our target is to estimate the effort needed to correct them.

1) The Estimation Method: Using the Delphi method [16] first the estimates were collected on how long a single instance of a given code smell type correction would take.

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

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

- Easy: The issue has only local effects if any, the context tells the original intent and there is no need to change external systems[4].
- 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[5].

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.

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

2) Estimation Results: We summarize the results in Table I.

3) Analysis of Estimation: 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 estimation 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.

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

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

[5]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.

TABLE I
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
visbility 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

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

B. The Cost of Fixing Standardized Test Suites

Applying the estimated correction times we were able to calculate the technical debt of both 3GPP[6] and ETSI projects (Table II).

We found that standardized test suites have substantial technical debt.

In the average difficulty case[7], the technical debt of the projects can be measured on 1000 Mhr base. Meaning several man-years of technical debt.

C. Validity

This study was performed with a small group of experts working at the same company. This study might suffer from the usual threats to external validity. There might be limits

[6]3rd Generation Partnership Project

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

TABLE II
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

to generalizing our results beyond our settings (programming language and tools used, project setups, experience of the experts and possible industry specific effects).

Some of the projects contained syntactical and semantical errors ([2]). 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 * in Table II, import modules of non TTCN-3 or ASN.1 kinds. These are not supported currently by our tool or have incomplete archives. In those modules the correct number of the founded issues could be higher.

IV. CONCLUSION

In our ongoing research activities we try to understand how test systems can be viewed as software systems, whether there are any differences. In previous articles we have shown that large test systems can grow into complex structures ([1]). We have also shown ([2]) that standardized test suites might contain a large number of code smell instances.

In this article we set out to connect our research results with industrial projects by showing the cost of fixing technical issues in the investigated test systems.

Applying the Delphi method we

- estimated the cost of fixing TTCN-3 code smells,
- calculated the technical debt of several standardized test suites.

We found that the existing standardized test suites contain substantial technical debt. Following the results in [7], assuming that test systems follow the same rules as software systems, these numbers should be reduced to secure higher customer satisfaction[8] on the long term.

We also observed that, according to our experts, in TTCN-3 the cost of fixing a code smell instance of average difficulty is very close to the easiest case.

V. FURTHER WORK

In this article we exposed that the technical debt in the examined project could take years to pay back completely. This creates a valid case for further study, on the possible automation of the correction process.

ACKNOWLEDGEMENTS

The authors would like to thank the DUCN Software Technology unit of Ericsson AB, Sweden for the financial support of this research and the Test Competence Center of Ericsson Hungary for providing access to their in-house tools. These proved to be invaluable to our measurements.

We would like to thank András Rókás, János Sváner, Attila Kovac, Rolf Deme, László Zeke, Eduárd Czimbalmos, Jenő Balaskó, János Kövesdi, Sándor Juhász, Gergely Nagy for their help in creating the estimations.

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] 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, 2009.
ISBN: 978-3-642-05030-5 DOI: 10.1007/978-3-642-05031-2_19

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

[8]Customers of test systems can be the organizations themselves using the systems verifying their products. Issues with the technical quality of tests can result in ambiguous or misleading test results and might set limitations on what can be tested.

DOI: 10.14794/ICAI.9.2014.2.273

[3] 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, pages 29-30.
DOI: 10.1145/157710.157715

[4] 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), pages 19-26.
DOI: 10.1109/MTD.2014.13

[5] 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), pages 35-42.
DOI: 10.1109/MTD.2014.8

[6] J. Ho, G. Ruhe, *When-to-release decisions in consideration of technical debt*, in 6th International Workshop on Managing Technical Debt (MTD), pages 31-35.
DOI: 10.1109/MTD.2014.10

[7] N. Ramasubbu, C.F. Kemerer, *Managing Technical Debt in Enterprise Software Packages*, 2014, IEEE Transactions on Software Engineering, Volume 40, Issue 8, pages 758-772.
ISSN: 0098-5589 DOI: 10.1109/TSE.2014.2327027

[8] Z. Li, P. Avgeroiu, P. Liang, *A systematic mapping study on technical debt and its management*, 12/2014, Journal of Systems and Software, Volume 101, pages 193-220.
DOI:10.1016/j.jss.2014.12.027

[9] M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

[10] F. Khomh, M.D. Penta, Y.G. Guhneuc, *An Exploratory Study of the Impact of Code Smells on Software Change-proneness*, 2009, WCRE (75-84). IEEE Computer Society.

[11] 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, ESEM 2009, pages 390-400.

[12] 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*, 2008, Balancing Agility and Formalism in Software Engineering, pages 252-266.
ISBN: 978-3-540-85278-0 DOI: 10.1007/978-3-540-85279-7_20

[13] A. Yamashita and L. Moonen, *Do developers care about code smells? An exploratory survey*, 20th Working Conference on Reverse Engineering (WCRE), 2013, pages 242-251.
DOI: 10.1109/WCRE.2013.6671299

[14] A. Yamashita and L. Moonen, *Do code smells reflect important maintainability aspects?*, 28th IEEE International Conference on Software Maintenance (ICSM), 2012, pages 306-315.
ISSN: 1063-6773 DOI: 10.1109/ICSM.2012.6405287

[15] A. Kovács and K. Szabados, *Test software quality issues and connections to international standards* in Acta Universitatis Sapientiae, Informatica, 5/2013. pages 77-102.
ISSN 1844-6086 DOI: 10.2478/ausi-2014-0006

[16] L. Helmer, *Analysis of the future: The Delphi method*, March 1967, RAND Corporation http://www.rand.org/pubs/papers/P3558.html