Measuring and Improving Testability of System Requirements in an Industrial Context by Applying the Goal Question Metric Approach

Armin Beer
Beer Test Consulting
Baden bei Wien
Austria
armin.beer@bva.at

Michael Felderer
University of Innsbruck, Austria
Blekinge Institute of Technology,
Sweden
michael.felderer@uibk.ac.at

ABSTRACT

Testing is subject to two basic constraints, namely cost and quality. The cost depends on the efficiency of the testing activities as well as their quality and testability. The author's practical experience in large-scale systems shows that if the requirements are adapted iteratively or the architecture is altered, testability decreases. However, what is often lacking is a root cause analysis of the testability degradations and the introduction of improvement measures during software development. In order to introduce agile practices in the rigid strategy of the V-model, good testability of software artifacts is vital. So testability is also the bridgehead towards agility. In this paper, we report on a case study in which we measure and improve testability on the basis of the Goal Question Metric Approach.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

System testing, requirements-based testing, testability, complexity, requirements quality, traceability, software quality, empirical study.

*Produces the permission block, and copyright information
†The full version of the author's guide is available as a smart.pdf document
[1]It is a datatype

RET'18, June 2, Gothenburg, Sweden

https://doi.org/10.1145/123_4

1 INTRODUCTION

Requirements-based testing has been recognized as the key to aligning business value and risks in industry [22], [23]. Systems in the social insurance domain are tested on the basis of the V-model. The test cases are developed from the artifacts of the requirements specification. For instance, business rules and a specification of the user interface are used in the system testing of use-case descriptions. The long duration of projects and frequent changes of the requirements mean that test cases have to be changed. In order to introduce agile practices into the rigid strategy of the V-model, the software artifacts must be easily testable [24].

In this paper, we report on a case study for applying a methodology to measure and improve testability in large-scale projects, taking frequent requirement changes into account. We performed a retrospective analysis of three iterations on the basis of an analysis of complexity, release planning, effort, time, defect data and experience. The aim was to show that investments in testability can promote improvements in future iterations. Improvements are feasible in terms of (1) reduction of complexity of the requirements artifacts, (2) balancing development and test effort, and (3) better final quality.

This paper is organized in the following way: Section 2 describes the terms and background of testability. Section 3 covers the study design. Section 4 presents the industrial context. Section 5 presents the cases and Section 6 the application of the GQM (Goal Question Metric) method. Section 7 then reports on the results and Section 8 closes with a conclusion and a review of future work.

2 BACKGROUND AND RELATED WORK

2.1 Testability

Software testability [1] is the degree to which a software artifact supports testing. If the testability of the software is high, it is easier to find faults in the system by means of testing. A lower degree of testability results in an increased test effort [8]. Figure 1

shows the relationship between the complexity and testability of a Software Requirements Specification (SRS) or architecture on the one hand and the components on the other hand. If the complexity of the SRS is high, for instance, the testability and the maintainability of the test cases are low. For good maintainability of test cases, the testers have to understand the relationship between the different artifacts and the impact of changes, for instance in the legacy requirements.

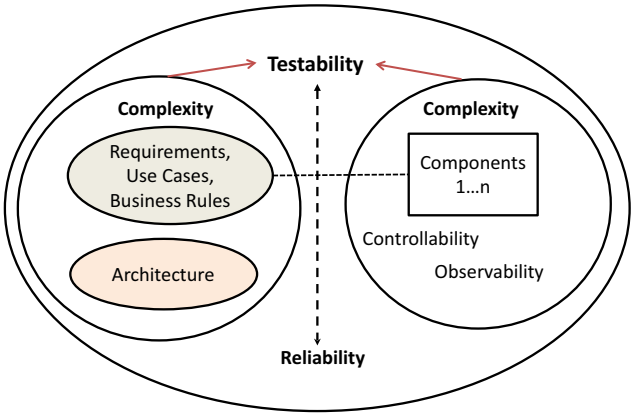

Figure 1: Testability of software systems

We will now define complexity and focus on the complexity of the SRS.

2.2 Complexity

A system is classified as complex if its design is unsuitable for the application of exhaustive simulations and tests, so that its behavior cannot be verified by exhaustive testing [5]. Uncontrolled complexity in software may lead to higher costs, less rigorous testing or reduced performance (see also NASA report [7]). The IEEE Standard Computer Dictionary defines complexity as the degree to which a system or component has a design or implementation that is difficult to understand and verify [6]. This phrase suggests that complexity is relative to the observer, so that that a skilled analyst, for example, would be able to analyze and design a well-structured system.

Egyed [14] lists the various categories of complexity that have to be taken into account when requirements are changed:

- Many-to-many mappings, for instance as requirements for design elements
- Incompleteness and inconsistencies
- Different stakeholders in charge of different software artifacts
- An increasingly rapid change of pace
- A non-linear increase in the number of software artifacts during the course of the SW life cycle (n^2 complexity).

Kan [9] presents complexity metrics to provide clues for software engineers in order to improve the quality of their work. Cyclomatic complexity (McCabe 1976) was designed in order to indicate program testability and maintainability. If an organization can establish a significant correlation between complexity and defect level, then the McCabe index can be a useful help to:

- identify complex parts of code needing detailed inspection
- identify non-complex parts likely to have a low defect rate
- estimate the programming and testing effort.

Card and Glass [9] developed a system complexity model which is a sum of structural (inter-module) complexity and overall data (intra-module) complexity. They found that the measure of system complexity was significantly correlated with a subjective quality assessment by a development manager and with the development error rate. They also provide guidelines to achieve a low-complexity design. The high correlation between module changes and enhancements illustrates the fact that as the number of changes increases, so do the chances for injecting defects. Small changes are especially error-prone. Kan [9] concludes that the key to achieving quality is to reduce the complexity of software design and implementation in a given problem domain.

2.3 Reliability

The lack of testability and the high complexity of the SRS influence the efficiency of test-case design and consequently the reliability of a product. Grottke and Dussa-Zieger [10] relate the test case coverage to the number of failures experienced.

FMEA (Failure Mode and Effect Analysis) is a bottom-up analysis designed to identify potential failure modes with their causes. Software Failure Mode and Effect Analysis (SFMEA) takes into account the complexity of system features in order to assess a Risk-Priority Number (RPN) as well as to define the test strength [13] and guide the test case design. In order to mitigate any risks of lack of testability, high complexity and system unreliability, the technical debt metaphor promotes a root cause analysis of deficiencies in the analysis and testing process and the introduction of a measurement program in industry.

2.4 Technical debt

Alves et al. [11] identified the following forms of technical debt:

- Architectural debt: high complexity of the overall system architecture
- Defect debt: for instance defects not fixed due to a lack of resources or low prioritization
- Design debt: code that violates good design practices
- Requirements debt: for instance bad testability of requirements
- Test automation debt: for instance bad testability of the system under test.

2.5 Goal Question Metric (GQM)

To improve the software development process, suitable goals have to be defined. These goals should support the business objectives and take the actual status of ongoing projects into account. The Goal Question Metric method of Basili [17] and van Solingen [20] provides an efficient framework for the improvement of development and testing activities and its approval by a project manager, for instance.

The GQM method contains the phases of planning, definition, data collection and interpretation. The definition phase identifies a goal such as the improvement of testability, and questions the related metrics and assessments. During the data collection phase, collection forms such as EXCEL templates are defined and project data, for instance relating to the defect tracking tool, are entered. During the interpretation phase, the measurements are used to answer the stated questions. These answers are again used

to see if the stated goals, such as an improvement in release quality, are achieved.

The case study follows the guidelines of Per Runeson and Martin Höst [21] in executing the following steps: (1) Case study design, (2) preparation of data collection, (3) collecting evidence, (4) analysis of collected data, and (5) reporting. The GQM method is applied to introduce a measurement program and to mitigate any technical debt encountered in the mentioned cases.

3 CASE STUDY DESIGN

This section presents the design of the study. The core content of our research is to investigate the effects of a lack of software testability with special focus on requirements-based testing from a retrospective perspective. We evaluate two projects quantitatively, and document the lessons learned.

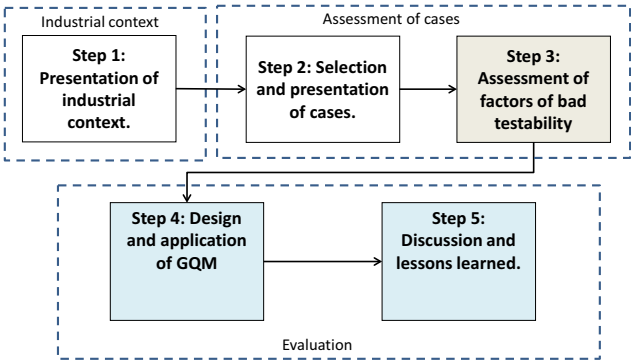

Figure 2: Design of the study

Step 1: We present the industrial context, taking testability problems into account

Step 2: We present the selected projects of a social insurance institution, which are assessed in the next step

Step 3: Testability degradations are observed in the mentioned cases. We assess the reasons for bad testability.

Step 4: We define metrics to monitor the test process and perform an evaluation of two projects, applying the GQM (Goal Question Metric)

Step 5: We discuss the results and improvement measures in order to enhance testability and reduce the complexity of software artifacts.

4 INDUSTRIAL CONTEXT: STEP 1

In the studied institution, a standard development and testing process based on the ISTQB standard has been in place for several years. Several web applications aiming to automate the business processes of the institution, for instance refunding invoices for the medical care of insurants, are developed. New external components were added to the core system so that the complexity of this system of systems increased during the last few years [22]. Good testability of software artifacts is therefore a key issue in order to deliver applications of high quality within time and budget. In the studied cases, testability is influenced by the growing complexity of software artifacts, for example by change requests, new interfaces, platform upgrades etc. In the case study presented by Felderer and Ramler [15], additional modules which introduced further inter-module dependencies constituted a major complexity driver. Jungmayr [16] presents an approach to define the metrics of software dependencies, namely ACD (Average Component Dependency), which takes the extent of the influence of the dependences in software design into account.

The selected cases were analyzed in order to find reasons for bad testability and their improvement by applying GQM.

5 SELECTION AND ASSESSMENT OF CASES: STEPS 2 and 3

Step 2 – Selection and presentation of cases

This section presents the cases selected for this study.

Case A:

- A new complex system for the management of subscriptions of insurants linked to software components of various institutions of the government, companies and the bank is under development
- Frequent requirements changes during development have an impact on testability
- Development of the software lasts about five years and the total effort is about 10,000 person days
- Complex performance requirements
- Synchronization with release plans of coupled systems.

The degradation of testability is recognized as a key problem in progressing the iterative development.

Case C:

- This is a core application concerned with charging the monetary obligations of insurants and the healthcare institution
- Analysts created the requirements and components of good testability at the start of the project
- New interfaces to external components were added during development.

Attributes	Case A	Case C
Goal	Managing the subscriptions of social insurants	Automating the business of social insurance
Duration of project [year]	4	7
Number iterations	10	4
Number of releases	25	8
Number of features	65	20
Number of reqs.	370	80
Iteration 3: test effort plan [PD]	217	215
Percentage of test effort to development effort	40%	30%
% of manual TCs to be automated	50%	50%

Table 1: Description of Cases

The two projects were selected after analyzing several candidates because the organization wanted to improve the efficiency of testing and introduce a measurement program. A second point was that these projects were under development at that time and the complexity of the software has increased from iteration to iteration.

Step 3 – Assessment of factors of bad testability in the social insurance institution

In order to handle the factors effecting testability encountered in projects of a social insurance institution, we used the testability fishbone visualization of Binder [19] and transformed it into a mind map (Figure 3). We will focus on process issues rather than on technical problems.

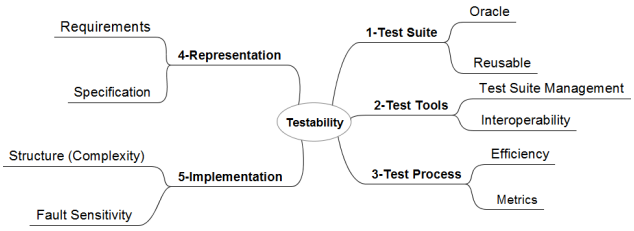

Figure 3: Testability factors in SW projects

1-Test suite: The test oracle is deduced from the requirements specification. Traceability should be in place in order to adapt test cases when the requirements are altered. Good testability is a prerequisite for the efficient design and execution of test cases.

2-Test tools: In system testing, the tools should support test suite management, test case development and reporting. To embrace changeability, the tools for analysis and testing should be tightly coupled.

3-Test process: The test process is based on the V-model. The effort and duration of the verification and validation phases lead to iterations of 3-4 months each. The number of detected bugs influences the degree of testability. Metrics to monitor the test process are missing or difficult to evaluate. The high effort of maintaining an automated test diminishes the return of investment from automation.

4-Representations: The usefulness and testability of representations constitute a critical factor of test-case design. In our case, natural language and semi-formal representations in UML are used. The specifications should contain relevant information for system testers on domain level as well as detailed technical descriptions for developers.

5-Implementation: The system architecture can be too complex: Structural (fan-in and fan-out) and system complexity (inter-module and overall data complexity) are not taken into account. The detectability and localization of faults and the effect of a failure may prove difficult. Permanent performance issues may hinder the development and execution of automated regression tests of the iterations.

6 EVALUATION: STEP 4: APPLICATION OF GQM.

To prevent degradation of testability during the development cycle, we need a framework and metrics. We apply the four phases of the goal-question-metric (GQM) approach of Basili [17] and van Solingen [20]: 1-planning, 2-definition, 3-data collection, 4-interpretation in order to improve the efficiency of testing.

6.1 Planning phase

In this phase, a framework is provided to introduce a measurement program. The starting point was to assess the current challenges and problems in ongoing projects and select the appropriate metrics. The relevant stakeholders, i.e. analysts, test managers and testers, are selected in order to have access to project documents such as test reports, effort estimation etc. Templates, for instance using the spreadsheet program EXCEL for the collection of data, were also prepared.

6.2 Definition phase: Definition of goals, questions and metrics

The primary goals for the test management are to keep the testing efficient, to reduce the time needed for testing and to improve the quality achieved by testing. These goals can be refined in terms of more specific sub-goals in order to improve testability:

G1. Requirement artifacts should be testable and support the viewpoint of developers and testers

G2: The quality of the releases or iterations should continually improve

G3: Change requests should have a low impact on architecture and test cases

G4: The effort of localization, correction and retesting of software defects should be minimized

G5: The ROI of test automation should increase during the testing of releases.

G1. Requirement artifacts should be testable and support the viewpoint of developers and testers.

Q1: What is the complexity of the requirements to be tested?
M1.1: Complexity of features and SRS

Q2: What is the testability of the requirements to be tested?
M2.1: Degree of testability
M2.2: Categories of requirement anomalies, i.e. the requirement anomalies detected by a review.

G2: The quality of the releases or iterations should continually improve

Q1: How can we assess the efficiency of test results?
M1.1 Test results and report

Q2: How can we assess the quality of releases?
M2.1 Defect trend analysis.

G3: Change requests should have a low impact on architecture and test cases

Q1: What are the consequences of change requests for testing?
M1.1 Number of regression test cases
M1.2 Test effort.

G4: The effort of localization, correction and retesting of software defects should be minimized

Q1: How can we assess the testability of the architecture.

M1.1 Testability of architecture and average component dependency

M2.1 Effort to analyze, fix and retest defects.

G5: The ROI of test automation should increase during the testing of releases

Q1: How can we measure the degree of automation of test cases in order to reduce the time of regression tests?

M1.1 Number and percentage of automated test cases.

6.3 Data collection and recording

The source of the data are test effort estimations, review results, minutes of meetings, protocols of the steering committee, test reports, the test case design and execution documented in a test management tool etc. The study data are recorded in spreadsheets as well as protocols of discussions with the analysts and test manager.

6.4 Analysis and interpretation of collected data

The goal of the measurement was to understand the root causes of the increase of testability problems from iteration to iteration and to find remedies in order to meet the deadline for shipping the applications. A second goal was to define a measurement program for future projects.

We will now present the application of the GQM method to the use cases.

G1. Requirement artifacts should be testable and support the viewpoint of developers and testers

Q1: What is the complexity of the SRS to be tested?

M1.1: Complexity of features and use cases.

The analyst and system architect assess the degree of complexity. The skill of the teams and the testability of the features or requirements are taken into account. The effort of analysis, development, architecture creation and testing can be calculated more precisely by taking these influencing factors into account. The relationships between the factors assigned to analysis, development etc. as depicted in Figure 4 are heuristic.

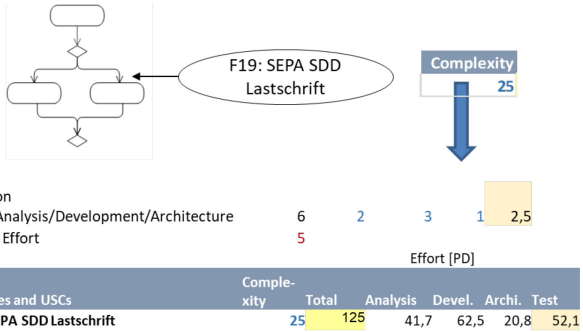

Figure 4: Case C: Calculation of complexity and effort of development

In case A, the complexity of the requirements artifacts is assessed by assigning attributes (low and high only).

Complexity measures allow a realistic calculation of the effort of analysis, development, architecture creation and testing. This conclusion was based on the experience gained in case C.

Q2: What is the testability of the requirements to be tested?

M2.1: Degree of testability

The number of TCs not executed and the percentage of automation indicate the severity of the testability problems encountered. As shown in Table 2, only 17% of the TCs could be automated in iteration 3. In case C, about 37% of the TCs were automated. Feedback from domain experts, developers and testers of case C also indicated that the requirements specification is of good quality in terms of testability and ease of understanding.

Release	Passed	Failed	Not exec.	Number of manual and automat. TCs	Number of automat. TCs	% automated
Case-C Rel. V 3.8.4	1424	111	109	1644	607	36.92%
Case A Rel. V 0.8.0	1909	292	393	2594	441	17.00%
Case-C Rel. V 3.8.5	880	45	706	1631	603	36.97%
Case-A Rel. V 0.9.0	1390	178	947	2515	457	18.17%

Table 2: Degree of testability in case A and C

In case A, the number of change requests cumulated in iteration 3. It showed the importance of building-in testability already in the first iteration.

M2.2: Categories of requirement anomalies

Case A: The test management assessed the quality of the requirement specifications of iteration one and detected the categories of defects shown in Table 3. All testability issues had a high impact on the design of test cases.

Testability issue	Description	Example
Completeness	Use-case descriptions	Links in use case description are missing.
Defects	Business rules	Rules contradicting
Traceability	Mapping	Coverage of business processes
Comprehen-sibility	Natural language	Focus on the developer point of view, too technical for domain experts.
Right level of detail	For developers and tester	Sorting rules of items are not detailed enough to design test cases.

Table 3: Categories of anomalies of SRS detected by reviewers in iteration 1

Great effort is invested in reviewing the requirements. The example of case A is shown in the following table. The major and critical anomalies detected by the various reviewers appeared as

testability problems from the viewpoint of the testers (Table 4). Testability issues detected by the testers were regarded as problems for designing good test cases.

Iteration	Number of Reviewers	Number of anomalies	Category 2 (major) or 3 (crit4ical)
2c	11	269	2
3b	11	557	10
3d	7	641	8
3	10	219	8

Table 4: Number of anomalies of SRS detected by reviewers

G2: The quality of the releases or iterations should continually improve

Q1: How can we assess the efficiency of test results?
M1.1 Test results and report

Case A: In Rel. 0.6.0, 1,782 test cases were executed and about 20% failed because of requirement changes of an external component and schedule, and 30% of the planned TCs could not be executed. In Rel. 0.7.5, the database had to be redesigned and the test cases also had to be changed. Insufficient performance and interface defects explained why 30% of the detected defects had been "critical". This result indicates that the more changes there are, the more defects will occur [9].

In Rel 0.8.0: despite the "feature freeze" the number of open bugs still increased because the localization of defects was difficult and the testability was bad. The development resources were insufficient to correct the number of bugs in time.

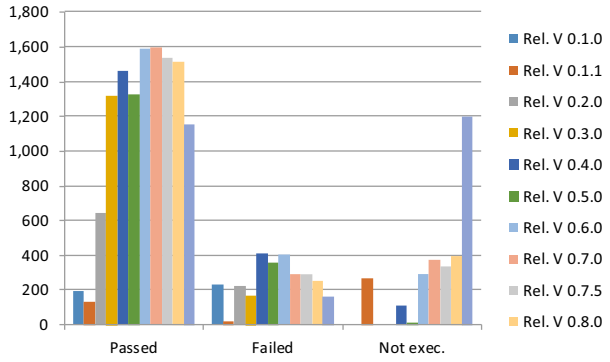

Figure 5: Case A: Result of system tests of releases

Case A: Figure 5 shows the test progress starting with the first iteration. The number of not-executed TCs increased in the last iteration before shipment because of still latent testability issues.

Q2: How can we assess the quality of releases?
M2.1 Defect trend analysis.

The defect analysis of case A shows that the total number of open bugs (not fixed or not retested) still increased when testing iteration 3. The number of open bugs is still increasing.

G3: Change requests should have a low impact on architecture and test cases

Q1: What are the consequences of change requests to testing?

One major challenge is the synchronization of the release plans of cases A and C during the last iteration before shipment. For example, the backlog of uncorrected defects and performance problems had an impact on testing in case C.

M1.1 Number of regression test cases

Testability problems cumulated in Case A Rel. V 0.8.0, because of the following issues:

- Complexity of the requirements specification increased compared to iterations 1 and 2
- Almost the complete set of test cases had to be changed because of numerous change requests
- 28% of the TCs could not be retested because of testability problems after bug fixing
- 35% of test cases failed because of testability problems.

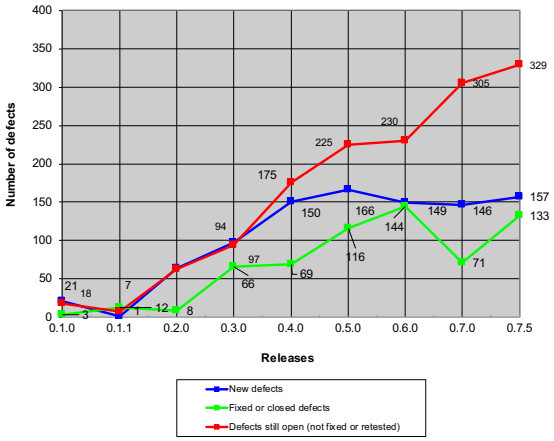

Figure 6: Defect trend analysis of Case A

Test case type	Number of Designed Test Cases	Number of Test Cases Executed in V 0.8.0	% of Test Cases not Executed
Regression tests	400	887	6.00%
Test data initialisation	190	557	13.50%
New features	405	346	29.48%
Retest of defects	100	363	28.10%
Automated tests	150	441	9.26%
Update of TCs	640		
Total	**1885**	**2594**	

Table 5: Percentage of TCs not executed in Case A Rel. V 0.8.0

M1.2 Test effort

The test effort is estimated taking the testability of the requirements into account and differs from project to project. For the design of test cases, 1.5 hours per test case is estimated in case A and 1.25 hours in case C. The execution of a manual test case is calculated by assuming 0.5 hours per test case. The test automation is performed by a separate team and the effort is calculated separately.

G4: The effort of localization, correction and retesting of software defects should be minimized

Q1: How can we assess the testability of the architecture?

M1.1 Testability of architecture and average component dependency.

In case C, the architecture took potential changes of interfaces or the connection to new components into account. Component dependencies are low.

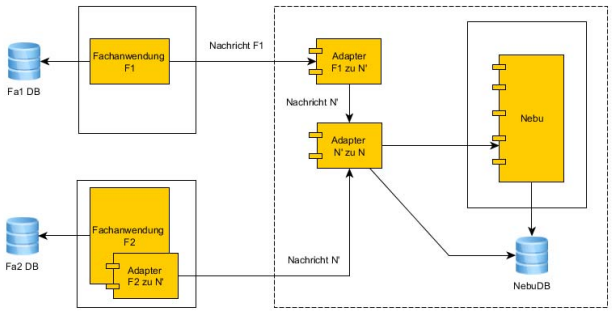

Figure 7: Case C testable architecture

M2.1 Effort to analyze, fix and retest one defect

Case A: The localization, correction and testing of defects is labor-intensive and takes about one day per defect, depending on the experience of the test manager. The main reasons for this are the poor testability of the software under test, the high complexity of the architecture (many interconnected components) and the complex relations between distributed data sets which are changed from iteration to iteration.

G5: The ROI of test automation should increase during the testing of releases

Q1: How can we measure the degree of automation of test cases in order to reduce the time of regression tests?

M1.1 Number and percentage of automated test cases.

Case A: The goal to automate 50% of the test cases could not be reached in case A because of frequent changes in use cases and masks. Only 17% of the planned automated test cases were automated in iteration 3.

6.5 Threats to validity

In this section, we present the threats to validity as well as suitable countermeasures [21]. These threats to construct validity are reduced by cooperation with the industry partner for about eight years. The consistency of the results gained from retrospective analysis, i.e. measures of testability in two projects, was assured by presenting the outcome to the test management of the social insurance institution for review. The data were collected by the consultant who was directly involved. The researcher reviewed the data. Further threats to internal validity were reduced by using established tools for reporting and defect management. Finally, the threats to external validity are reduced by conducting the case study in a real-world industrial setting with practitioners from different software companies.

Iteration	Case A Versions	Total Number of Test Cases	% Test Cases Automated
1	V0.1.0	420	no
	V0.1.1	420	
	V0.2.0	868	
	V0.3.0	1,489	
2	V0.4.0	2,117	6.38
	V0.5.0	1,779	4.72
	V0.6.0c	2,493	8.46
3	V0.7.0	2,258	no
	V0.7.5	2,154	
	V0.8.0	2,594	17.00
	V0.9.0	3,020	16.66

Table 6: Percentage of executed automated TCs in Case A

7 EVALUATION: STEP 5: DISCUSSION AND LESSONS LEARNED

In the interpretation phase, we will draw conclusions from the results of the study. These results are discussed in feedback sessions with the test manager. Presentation slides are prepared to focus on the main findings. The study concludes that the test manager should be involved during the analysis and design phase with the goal of reducing complexity and enhancing testability in order to mitigate unforeseen changes in the implementation and testing of future changes. Metrics should be used to monitor and promote continuous improvement of the development and testing procedures. We will now discuss these improvement measures, taking the interpretation of the results of the GQM application into account:

(1) Reduction of complexity and enhancement of testability

Requirements or platforms are changed and new interfaces are added in an iterative development process. As a consequence, the complexity of the software artifacts increases and testability is worsened. In black-box testing, the observability of the subsystems must be ensured in order to implement effective test cases. In case A, however, the subsystems cannot be accessed to verify the test results. In this case, the relatively simple top-level requirements hide the immense complexity introduced by legacy and a dependency on the components of a networked system. Analysts should therefore follow the guidelines in order to mitigate complexity and focus on testability. In case C, for example, the analysts added examples of abstract test cases to business rules and enhanced their testability by sign-pointing changes with colors. In case C, the testability improvements introduced by the analyst early in the analysis phase were accepted by the domain expert, the testers and the project management. The comparison of the percentage of automatic regression tests of adjacent releases of cases A and C shown in Table 6 indicates better testability in case C. Due to testability

problems, such as delays in test case design and uncorrected defects, the number of untested test cases in case A is high, as shown in Table 5.

(2) Working packages of testers
Working packages assigned to testers should be elaborated within time and budget. In case A, however, with testers of skills 0 compared to case C, the completion of working packages was delayed by a lack of understandability of the requirements and observability of the software under test. In case A, the effort of fixing and retesting defects required one day per defect because the fault localization needed frequent communication between developer and tester. In estimating the effort and duration of the working packages, the degree of testability has to be taken into account. An assessment of the complexity, as shown in Figure 4 improves the definition of the WPs.

(3) Frequent change requests
Change requests may have an impact on a large number of test cases. Traceability from a test case to the requirement artifacts must therefore be in place. The tools of the analysts and test management should be tightly coupled. Alternative release plans should be taken into account [18].

(4) Efficient test management
The metrics presented in Section VI should be taken into account when monitoring a test project. Bug fixing and retesting should have priority.

(5) Shorter iteration and release cycles
In order to obtain earlier feedback, the current duration of a release implementation of about 6 to 12 months should be reduced.

(6) Good testability of the system architecture
The long duration of projects and the networking of different systems present a challenge to the system architect. The testability of the architecture, components and data has to be taken into account at the start of the project. The analyst of case C has implemented a testable architecture which takes the coupling of new components into account.

8 CONCLUSIONS

By comparing the results of a retrospective analysis of two projects, we have demonstrated that using metrics relating to complexity, testability and quality offers the following advantages: complexity of the SRS is reduced, good testability is recognized as important for efficient testing and more attention is paid to quality. These results are a confirmation and follow-up of earlier contributions, for instance on using defect taxonomies for requirements validation [22] or quality-driven release planning [18].

To sum up, our study indicates that application of the GQM method allows specific quality attributes to be monitored in order to create a framework for the improvement of testability. Experience in two large-scale projects under development shows that good testability is key for efficient development and testing of a software product. In the future, we will carry out further empirical studies and refine our framework for measuring testability based on the Goal Question Metric approach. Furthermore, we will also consider testability of non-functional properties.

ACKNOWLEDGMENTS

The paper was partly funded by the Knowledge Foundation (KKS) of Sweden through Project 20130085: Testing of Critical System Characteristics (TOCSYC).

REFERENCES

[1] ISTQB. Standard glossary of terms used in software testing, Version 3.1, Internat. Software Testing Qualifications Board, Glossary Working Party, 2016.
[2] A. Beer, R. Ramler, The role of experience in software testing practice, Proceedings of the 34th Euromicro Conf. Softw. Eng. and Advanced Applications: 258-265, IEEE 2008.
[3] M. Felderer, A. Beer, B.Peischl, "On the role of defect taxonomy types for testing requirements: Results of a Controlled Experiment", Proceedings of the 40th Euromicro Conf. Softw. Eng. and Advanced Applications, pp. 377-384, IEEE 2014.
[4] H. Femmer et al., "Rapid requirements checks with requirement smells", Journal of Systems and Software, Elsevier 2017.
[5] Defense Standard 00-54, Requirements for safety related electronic hardware in defense equipment, UK Ministry of Defense, 1999.
[6] IEEE Standard Computer Dictionary, 1990.
[7] Dvorak D.L., Eds., NASA study on flight software complexity, Jet Propulsion Laboratory, California Institute of Technology, 2001.
[8] Wikipedia, retrieval Dec.28, 2018
[9] Kan St., Metrics and models in software quality engineering, Addison-Wesley, pp. 311-330, 2006.
[10] Grottke M., Dussa-Zieger K.; Prediction of software failures based on systematic testing, EuroSTAR, Stockholm, 2001.
[11] Alves,N. S. R., Ribeiro L.F., Caires V., Mendes T. S., Spinola R. O., Towards an ontology of terms of technical debt. Proc. IEEE 6th Int. Workshop on Managing Technical Debt, pp 1-7, 2014.
[12] IEEE Standard Computer Dictionary, 1990.
[13] Sulaman M.S., et al, Comparison of the FMEA and STPA safety analysis methods – a case study, Softw. Quality Journal, Springer, 2017. (online first)
[14] Egyed A.; Tailoring Software Traceability to value-based needs. In Biffl St. et al. (Eds.): Value-based software engineering, pp.287-308. Springer, 2010.
[15] Felderer M., Ramler R., A multiple case-study on risk-based testing in industry, Int. Journal on Software Tools for Technology Transfer 16(5), Springer pp 609-625, 2014.
[16] Jungmayr S., Testability measurement and software dependencies, Proceedings of the 12th International Workshop on Software Measurement. Vol. 25. No. 9. 2002.
[17] Basili V.R., Rombach H.D., "The TAME project: Towards improvement-oriented software environments", IEEE Trans. on SW Engineering, Vol.SE-11, pp 758-773, 1988.
[18] Felderer M., Beer A., Ho J., Ruhe G., Industrial evaluation of the impact of quality driven release planning, ESEM '14, Torino, Italy, 2014.
[19] Binder R.V., Design for testability in object-oriented systems, Comm. of the ACM, Sept. 1994, Vol.37, No.9, pp 89-101, 1994.
[20] Van Solingen R, Berghout E., The goal/question metric method: a practical guide for quality improvement of software development; McGraw Hill, London, 1999.
[21] Runeson P, Höst M., Guidelines for conducting and reporting case study research in software engineering; Empirical Softw. Engg. 14,2 (April 2009), 131-164, 2009.
[22] Felderer M., Beer A., Using defect taxonomies for requirements validation in industrial projects; RE 2013, IEEE 2013, 296-301, 2013.
[23] Bjarnason, E., Runeson, P., Borg, M., Unterkalmsteiner, M., Engström, E., Regnell, B., Sabaliauskaite, G., Loconsole, A., Gorschek, T., Feldt, R., Challenges and practices in aligning requirements with verification and validation: a case study of six companies, Empirical Software Engineering, 19(6), 1809-1855, 2014.
[24] Garousi, V., Felderer, M., Kilicaslan, F.N., What we know about software testability: a survey, 2018, arxiv, CoRR abs/1801.02201