Master Thesis
Software Engineering
Thesis no: MSE-2009:14
September 2009

A Software Verification & Validation Management Framework for the Space Industry

Jan Schulte

School of Engineering
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

This thesis is submitted to the School of Engineering at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Software Engineering. The thesis is equivalent to 20 weeks of full time studies.

Contact information

Author:

Jan Schulte
jan@janschulte.com
Internet: www.janschulte.com
Phone: +49 6731 4825 912

University supervisor:

Dr. Robert Feldt
robert.feldt@bth.se

Software Engineering Research Lab
School of Engineering
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

Internet: www.bth.se/tek
Phone: +46 457 38 50 00
Fax: +46 457 271 25

Industrial supervisor:

Annalena Johansson
annalena.johansson@space.se

RUAG Aerospace Sweden AB
Solhusgatan 11
SE – 405 15 Göteborg
Sweden

Internet: www.space.se
Phone : +46 31 735 00 00
Fax: +46 31 735 40 00

University supervisor (co-supervisor)

Dr. Natalia Juristo
natalia@fi.upm.es

Empirical Software Engineering Research Group
Facultad de Informática
Campus de Montegancedo
Universidad Politécnica de Madrid
28660 Boadilla del Monte (Madrid)
Spain

Internet: www.fi.upm.es
Phone: +34 913367399
Fax: +34 913367412

ABSTRACT

Software for space applications has special requirements in terms of reliability and dependability. As the verification & validation activities (VAs) of these software systems account for more than 50% of the development effort and the industry is faced with political and market pressure to deliver software faster and cheaper, new ways need to be established to reduce this verification & validation effort.

In a research project together with RUAG Aerospace Sweden AB and the Swedish Space Corporation, the Blekinge Tekniska Högskola is trying to find out how to optimize the VAs with respect to effectiveness and efficiency.

The goal of this thesis is therefore to develop a coherent framework for the management and optimization of verification & validation activities (VAMOS) and is evaluated at the RUAG Aerospace Sweden AB in Göteborg.

Keywords: verification & validation, space industry, process improvement

ACKNOWLEDGMENTS

This thesis would not have been possible without the help of several people and I want to use this place to give my acknowledgments to them.

First and foremost I would like to thank my supervisor Dr. Robert Feldt for making this thesis possible and helping me throughout the whole thesis by contributing with ideas and feedback.

Also, I want to thank the whole software development team at RUAG Aerospace Sweden AB and in particular Annalena Johansson and Joakim Ekwall for their confidence in me and their unrestrained support and fruitful contributions.

Special thanks go also to the software development team at the Swedish Space Corporation and specifically to Tanja Nordebäck who made it possible to evaluate this work at SSC.

Furthermore I want to express my gratitude to Philip Preissing, who agreed on doing this thesis project with me without hesitation and took the burden to move to Göteborg in his final year of studies at the TU München.

My final thanks go to my parents, who made it possible for me to study and always support me whenever I need help.

DECLARATION OF ORIGINALITY

I assure the single handed composition of this master’s thesis only supported by declared resources.

Alzey, Germany, 2009-09-01

[illegible]

Jan Schulte

CONTENTS

1 INTRODUCTION

1.1 Software In Space

Software for space applications has special requirements in terms of reliability and dependability. Since existing software engineering standards do not to cope with these challenges [FEL09], the European Cooperation for Space Standardization (ECSS) has developed a set of standards to address these issues.

As any other software-related industry, the space industry is faced with political and market pressure to deliver software at lower cost and faster time-to-market while keeping up high the quality requirements [FEL09].

In a research project together with RUAG Aerospace Sweden AB and the Swedish Space Corporation, the Blekinge Tekniska Högskola (BTH) is trying to optimize verification & validation activities (VAs) with respect to effectiveness and efficiency. The companies feel that they find the same defects with multiple VAs and therefore see a chance to reduce this redundancy without negatively impacting quality [AHM09] [RAZ09].

1.2 RUAG Aerospace Sweden AB

This thesis project was done at the RUAG Aerospace Sweden AB which was a subsidiary of Saab until it was acquired by the Swiss- & Germany-based company RUAG [RUA09]. RUAG specializes in highly reliable on-board satellite equipment such as computer systems, antennas, microwave electronics etc. Thereby, RUAG develops both the hardware and the software. The company is headquartered in Göteborg, Sweden and employs in total 360 people, of which about 30 work in the software unit.

Typically up to five projects are developed simultaneously in varying team sizes of about 10 people. The software projects range from 10 KLOC to 100 KLOC depending on the project and are developed mainly in C but with some low level parts written in assembler.

1.3 Swedish Space Corporation

The Swedish Space Corporation is a wholly state-owned company and has specialized itself on the development and operation of satellite systems, as well as rocket & balloon systems, launch services, flight test services and maritime surveillance.

SSC has about 600 employees in five locations throughout Sweden. The head office and engineering center is located in Solna near Stockholm and employs there ca 130 people.

The analysis took place in the satellite systems division, which develops attitude orbit & control systems, advanced On-board Data handling system units, on-board basic and application software, propulsion systems, and ground systems for testing and mission control. Thus, the division is operating in the same domain as RUAG.

1.4 Problem description

In the first part of the collaboration with RUAG and SSC, BTH has focused on analyzing the current situation of both companies. They examined the companies' way of work and the challenges they face regarding verification and validation [RAZ09] [FEL09]. A master thesis [AHM09], conducted within this project, found three main problems: faults 'slip' through between development phases, inappropriate selection of VAs, and the process and documentation requirements imposed by the ECSS standards.

The goals and expectations of RUAG were furthermore explored in an open interview with two representatives from the company, the head of software development and one project manager. The central questions of this interview were:

1. What are the biggest problems in each development phase after detailed design?
2. What are your expectations from introducing the framework?
3. Is the goal to reduce the number of VAs or the effort within each activity?
4. Would/could you change a VA?

The summarized results are as follows:

Since RUAG spends much effort on verifying and validating the software to comply with the ECSS standard and the high reliability demands in the Space industry and at the same time the market pressure grows to produce software cheaper and faster, RUAG strives to reduce the effort spent on its VAs while keeping up the high quality. Therefore, RUAG seeks to gain a deeper understanding of its VAs and conduct and manage them in a *more systematic approach*:

1. RUAG believes that faults slip through (FST) to later VAs when they should have been found earlier. The concept of FST, introduced by Damm et al. [DAM06], should therefore be adapted to the RUAG software development processes to gain more insight into the VAs carried out by the company.
2. Related to this problem is, that there is a lack of understanding which kinds of faults exist, where they are found and where they should have been found. Hence, RUAG's aim is to answer the question, whether they *should focus on finding a certain kind of fault in each VA.*
3. RUAG's experience is that code inspection is the most effective VA. Since problems in the HW/SW interface also cause severe problems, more tests, e.g. exploratory testing, in this field would be beneficial. However, RUAG has no means to show to their customers and/or ESA that certain VAs are more effective and focusing on these VAs would increase the quality.

Therefore, RUAG's expectations from introducing the framework are the development of a fault classification specific to RUAG and a process of collecting data, hence a *way to make it easier to measure the correct data.*

Another question RUAG would like to answer with the framework is which VA they should conduct when delivering a draft software, to show the customer a high-quality prototype of the software very early in the process.

Since RUAG is bound to the ECSS, it is neither possible for them to reduce the number of VAs nor to exchange one VA by another, which was considered one possibility to improve the VAs before conducting this interview. This means, that the framework should focus more on improving existing VAs than selecting VAs from a (hypothetical) set of VAs.

Thus, RUAG faces the problem of a fault slippage between the different development and testing phases and a lack of insight into the VAs. It is unclear which types of defects are found in each VA, what types the VAs should find and if there is an overlap between the VAs, i.e. whether different VAs find the same kind of defects. Hence, a further insight into the effectiveness and efficiency of the VAs is required to make a case on which VAs to focus and to find out which VAs need to be improved. Furthermore it is unclear which measurements need to be performed to quantify the problems and how the VAs can be improved using the collected data.

1.5 Background

Instead of focusing on a single verification and validation activity, research [LIT00][KIT98] and industry [KIK01] show that combining different VAs can be more efficient in finding defects. To keep up with the constant market pressure that is omnipresent in industry, further investigation is necessary to guide industry on how to select VAs to maximize the defect detection efficiency while minimizing the effort spent. Some initial work that has been done towards the creation of a comparison framework is presented in the following.

Wojcicki and Strooper present in [WOJ07] an iterative selection strategy (ISS) for verification & validation activities. The VAs are thereby firstly selected by their efficiency in order to maximize completeness, and secondly by the effort, i.e. the cost, they require. Based on the data collected while applying the technique, the selection is refined iteratively.
In [DAM06] the concept of Fault-Slip-Through (FST) is presented. In this approach, the faults found are categorized according to which phase they belong to, i.e. in which VA they should have been found. Based on these findings and the effort of the VA, the improvement potential for each activity is calculated.
Wagner [WAG06] proposes a more analytical approach. The defect detection techniques are thereby compared using an economical metric, namely the return on investment (ROI). The model also considers the effect of combining different defect detection techniques.

A framework for the comparison of testing activities and formal verification is presented in [BRA06]. However, this approach focuses more on studying the synergy and relationship of these two activities and doesn't give any advice on how this combination can be optimized.

Apart from the work that focuses on optimizing VAs in particular, frameworks have been proposed like QIP [BAS94], Six Sigma [TAY02] or the SEI measurement process [MCA93] which target process improvement in general. They typically follow an iterative process based on a problem description or analysis, a measurement phase, a root-cause analysis followed by an implementation phase.

There exists also a standard, namely IEEE 1061 [IEE1061], related to software metrics and measurement. It defines a methodology for establishing quality requirements and finding, measuring and validating appropriate software product and process metrics.

1.6 Contribution

This thesis contributes with a framework for the efficient management and optimization of verification & validation activities in real-world software development. It addresses the problems stated in section 1.4, i.e. to solve the problem of fault-slip-through and gain a more thorough understanding of the VAs with respect to their efficiency and effectiveness. To the best of our knowledge, currently no such framework exists that solves these problems.

1.6.1 Aims & objectives

Existing process improvement frameworks like QIP and Six Sigma but also the ISS rely on an iterative methodology to gradually evolve the processes. This methodology is based on the principle that software process improvement is a step-by-step process in an iterative lifecycle [FOW90] and that software engineers need post-mortem analysis in order to improve the construction processes of future projects [BAS88]. Therefore, the framework should also be based on an iterative cycle, in particular the cycle of ISS.

As DeMarco points out in [DMA86] *you cannot control what you cannot measure*. Therefore it is necessary to find out which measurements need to be done in order to support such a framework.

Because the framework is going to be used in an industry setting, it has to be as lightweight as possible and should only require a minimal set of changes to the existing processes. Therefore, the framework should allow for a parameterization to incorporate existing measurements of a company and to variably adjust the effort that has to be spent on the framework.

Hence, this thesis tries to answer the questions which measurements need to be done and how they can be combined into a coherent iterative framework that allows for the management and optimization of VAs. The detailed aims & objectives of the thesis are therefore as follows:

- Adapt FST and ISS to the development processes at RUAG Aerospace Sweden AB
- Creation of a framework that enables the comparison of the effectiveness of different VAs in eliminating different types of software defects and the comparison of the efficiency of different VAs.
- Validation of the framework at RUAG Aerospace Sweden AB
- Description of a minimal set of changes that RUAG needs to make to adapt to the framework
- A consideration of how the framework could be applied to the more agile development processes used at Swedish Space Corporation (SSC).

1.6.2 Research questions

To achieve the aforementioned aims & objectives, the following research questions need to be answered:

1. How can Fault-Slip-Through (FST) and Iterative Selection Strategy (ISS) be combined into a coherent framework?
2. Which alternatives exist to the aforementioned models?
3. Which measurements need to be done to support this framework?
4. How can the framework support parameterization to include existing measures and thus, minimize the measuring effort to be real-world applicable?
5. How can the framework be adapted to a real-world industry setting by only doing minimal changes to the company’s processes?
6. How does the framework perform in a real-world industry setting?
7. How can the framework be adapted to a more agile process?

Table 1 shows the mapping of these research questions to the aims of this thesis.

Aim	Research Question(s)
Adapt FST and ISS to the development processes at RUAG Aerospace Sweden AB	RQ1. How can Fault-Slip-Through (FST) and Iterative Selection Strategy (ISS) be combined into a coherent framework? RQ4. How can the framework support parameterization to include existing measures and thus, minimize the measuring effort to be real-world applicable? RQ5. How can the framework be adapted to a real-world industry setting by only doing minimal changes to the company's processes?
Creation of a framework that enables the comparison of the effectiveness of different VAs in eliminating different types of software defects and the comparison of the efficiency of different VAs.	RQ2. Which alternatives exist to the aforementioned models? RQ3. Which measurements need to be done to support this framework? RQ4. How can the framework support parameterization to include existing measures and thus, minimize the measuring effort to be real-world applicable?
Validation of the framework at RUAG Aerospace Sweden AB Description of a minimal set of changes that RUAG needs to make to adapt to the framework	RQ6. How does the framework perform in a real-world industry setting? RQ4. How can the framework support parameterization to include existing measures and thus, minimize the measuring effort to be real-world applicable? RQ5. How can the framework be adapted to a real-world industry setting by only doing minimal changes to the company's processes?
A consideration of how the framework could be applied to the more agile development processes used at Swedish Space Corporation (SSC).	RQ7. How can the framework be adapted to a more agile process?

Table 1: Mapping of research questions to the aims

1.6.3 Expected outcomes

- A description of the framework
- A process guideline for RUAG describing how the framework can be applied to their processes.
- Validation results gained by interviews and, if possible, by the data measured in a real project
- A list of recommendations for improving the RUAG development process based on the application of the framework
- A description on the suitability of applying the framework in a more agile environment

1.7 Research methodology

First, the process documents of RUAG will be analyzed to gain an in-depth knowledge of their software development processes. This includes roles, development phases, VAs used in each phase and metrics collected. Furthermore, the existing data and metrics collected for the different VAs will be investigated and evaluated with respect to their suitability for usage in the framework.

A literature survey is carried out to reveal the current state-of-the-art including which alternative models exist for the selection of the VAs and the measurement of these. Additionally, the results of the master thesis [AHM09] serve as input for providing preliminary data of the efficiency and effectiveness of the VAs in RUAG.

Based on these inputs (the previous and related work, the RUAG processes and existing measurement data) as well as the expectations and goals of RUAG from section 1.4, a coherent framework will be developed. It will be documented in a written framework description.
It is then determined whether the existing data is sufficient to support the application of the framework. According to the measurement levels, a minimum set of changes to the RUAG processes is developed. Those boil down to a process guideline describing what needs to be done to apply the framework at RUAG.

The framework will then be analyzed statically by reviews or if possible through to the application of the framework in a real project and analyzing the data obtained after a certain period of time. Based on the feedback gained, the framework will be revised iteratively.

Based on the framework and the evaluation a guideline is established giving improvement recommendations to RUAG.

In cooperation with SSC it is evaluated how the framework can be adapted to a more agile process using interviews or workshops. A description concerning the suitability is developed including possible necessary adaptations.

Figure 1 shows the research methodology as a workflow and describes in which steps the research questions are answered.

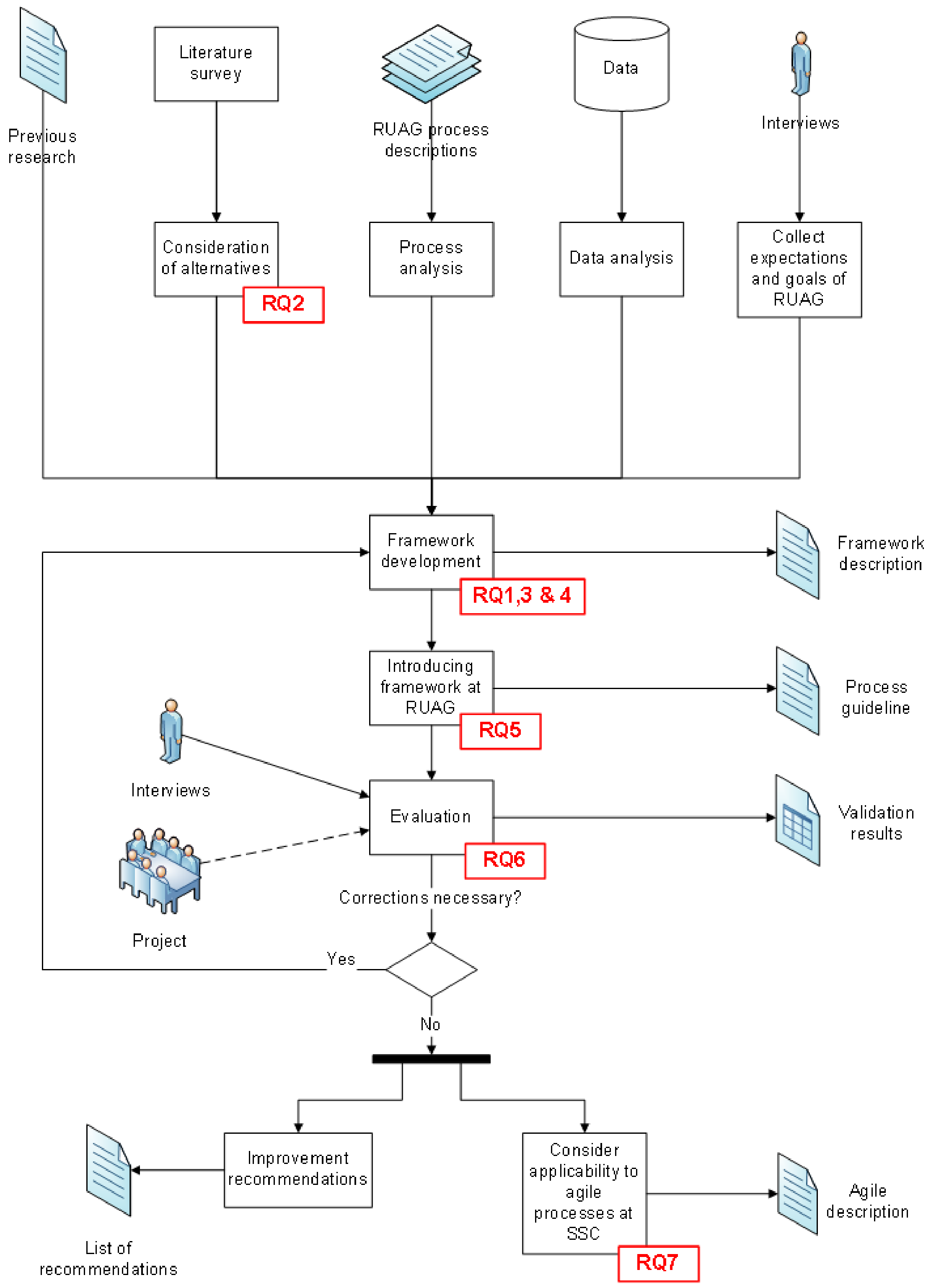


Figure 1: Research methodology

1.8 Thesis outline

The thesis begins with a detailed view of the software development process at RUAG in chapter 2 to gain a thorough understanding of the environment. In chapter 3 the framework is detailed and in chapter 4 its application to RUAG is described. Chapter 5 outlines how the framework could be applied in an agile environment at SSC and chapter 6 concludes this thesis with a discussion.

2 RUAG DEVELOPMENT PROCESS

2.1 ECSS Standard

The European Commission for Space Standardization has defined a set of standards for management, product assurance and engineering [JON97]. Since the customers demand RUAG to follow the ECSS standard, it is necessary to take a closer look at the standard in order to understand the RUAG software development process.

The ECSS parts related to software are the ECSS-E40 Part 1 and 2 for *software* [ECSE40-1] [ECSE40-2] as well as the ECSS-Q80 for *software product assurance* [ECSQ80]. The standard pays special attention to the requirements engineering, the architecture as well as verification & validation processes [ECSE40-1]. The standard follows a hierarchical customer-supplier model that means that the supplier of a product for a customer can in turn itself be a customer to a subcontractor.

The ECSS Software Lifecycle process is depicted in Figure 2 and shows the processes (the bars) and the reviews (the circles at the bottom) which have to be carried out to be ECSS compliant. The software is developed in the system engineering, the SW requirements & architecture engineering, the SW design & implementation engineering and the delivery & acceptance process. In parallel to this, supporting processes and verification & validation processes are performed.

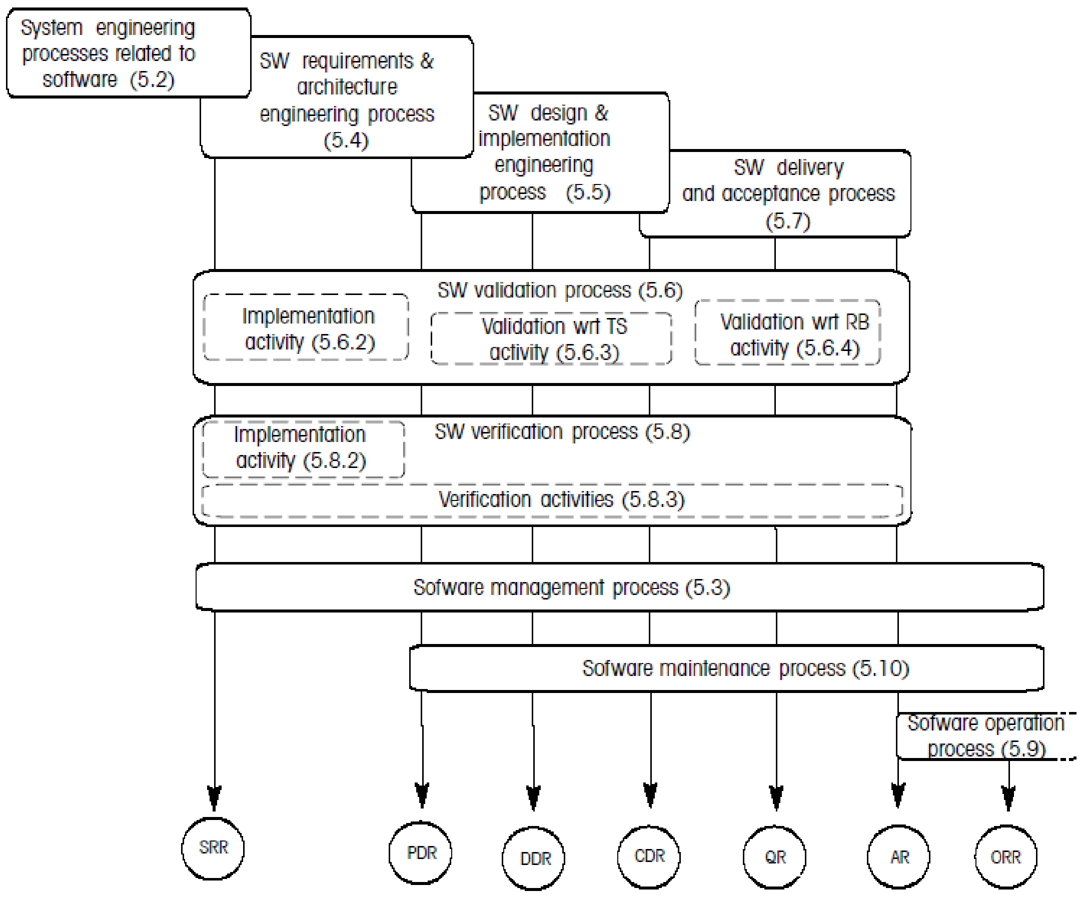

System engineering processes related to software (5.2)
SW requirements & architecture engineering process (5.4)
SW design & implementation engineering process (5.5)
SW delivery and acceptance process (5.7)
SW validation process (5.6)
Implementation activity (5.6.2)
Validation wrt TS activity (5.6.3)
Validation wrt RB activity (5.6.4)
Implementation activity (5.8.2)
SW verification process (5.8)
Verification activities (5.8.3)
Sofware management process (5.3)
Sofware maintenance process (5.10)
Sofware operation process (5.9)
SRR
PDR
DDR
CDR
QR
AR
ORR

Figure 2: ECSS Software Lifecycle Processes [ECSE40-1]

The reviews are the interaction points between the customer and the supplier and play therefore an important role in the ECSS standard. The need to be carried out at the end of the processes or sub processes and demand from the companies that documentation is available at these milestones that the processes and sub processes were carried out successfully. The full list of the reviews is shown in Table 2.

SRR	Software Requirements Review
PDR	Preliminary Design Review
DDR	Detailed Design Review
CDR	Critical Design Review
QR	Qualification Review
AR	Acceptance Review

Table 2: Reviews in the ECSS

The software requirements review (SRR) marks the beginning of the design phase and establishes the requirements baseline (RB) of the software development [ECSE40-1]. The supplier further elaborates the RB to a technical specification (TS).

Based on the requirements in the technical specification, the architecture and design is developed in the SW requirements & architecture engineering. This process is completed with the preliminary design review (PDR) and leaves the software development in the so called *specified state*.

After the PDR, the SW design & implementation engineering process is carried out in which the software is further designed and implemented. During this process, the supplier has to perform a unit test as well as an integration test. In the unit test the supplier is required to test all units against the requirements. In the integration test the developed aggregates are tested against the requirements. In parallel to this process, the validation against the technical specification has to be carried out.

The design activities end with the critical design review (CDR) and must be undertaken before the software may be integrated with higher level systems in the hierarchy. With the CDR the software project is in the *defined state* and after that review every problem has to be documented in a software problem report (SPR).

The CDR follows the SW delivery & acceptance and in parallel, the software has to be tested with respect to the requirements baseline. The next review is the qualification review (QR) that marks the end of all verification activities against the requirements baseline and leaves the project in a *qualified state*.

The last review performed is the acceptance review (AR) by the customer. The end of this review marks the software project as being in an *accepted state*.

Apart from these mandatory reviews, the SW design & implementation engineering process, i.e. the process between PDR and CDR, can optionally also include a detailed design review (DDR) and a test readiness review (TRR) if requested by the customer. Whereas the DDR is a review of the refined design and the TRR is conducted to assure that all test facilities and test cases are available at the start of the test campaign.

Additionally, the customer may require that further verification & validation is performed by an independent third party. These VAs are called independent software verification and validation (ISVV).

2.2 V-model

The ideal software development process at RUAG follows a classical V-model, depicted in Figure 3. On the left path the production phases are shown with the documents/artifacts that are produced and the right path shows the VAs that are carried out.

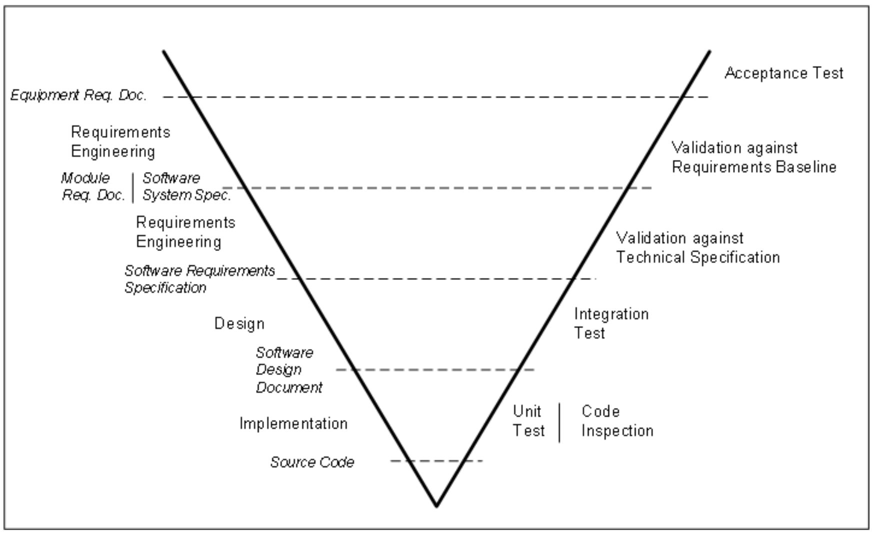


Figure 3: Software development process at RUAG

Since RUAG develops only hardware components as a whole and does not sell standalone software, the first part of the V-model contains both, hardware and software aspects.
The process starts with RUAG getting the requirements from the customer in a so-called *equipment requirements document* (ERD), which consists of the whole system requirements including hardware and software.

Based on this document, the *requirements baseline* (RB) is developed by RUAG, which consists of more detailed requirements and the separation into hard- and software modules (which is written down in the *module requirements document* (MRD)). The software modules are further specified in the *software system specification* (SSS), which describes the software.

With the SSS, the software-only part of the V-model starts. From this SSS the *technical specification* (TS) is developed (also called *software requirements specification* (SRS)), which is more detailed and solution oriented than the SSS. This additional step might seem unnecessary, but is a requirement imposed by the ECSS standard.

Based on the TS the design is carried out and a *software design description* (SDD) is developed, which further details the software system into units and is done using a modified (domain-specific) version of UML. Using this input, the software implementation phase starts and the source code is developed.

After implementation, the V&V path of the V-model starts. Theoretically it starts with *unit test* (UT). The implementation phase and the unit tests however are not clearly separated, since the source code is verified during implementation by the developer performing the unit test while implementing the software.

The following step up the V-model is the *code inspection* (CI). Thereby, the software is verified using static analysis tools and several in-house-developed scripts, as well as manual code inspection using checklists.

An *integration test* (IT) against the SDD follows the CI phase. This is a rather lightweight verification activity, since RUAG doesn't develop large-scale software systems with complex structures.

After IT, a *validation against the TS* (VT-TS) is performed. The software is thereby validated against the TS using the validation tests.

The final VA performed by RUAG is the *validation against the requirements baseline* (VT-RB).

The last step is the *acceptance test*, which is done either by the customer itself or by a third party, with the support of RUAG.

One distinctive characteristic of RUAG is that the software is always developed in two separate and independent teams, the *software design team* and *software validation team.* This means, that no member of the first team may take part in any activity of the second team and vice-versa. The responsibility of the software design team is to specify, develop and verify the software, while the software validation team's responsibility is to validate it.
The rationale behind this is to remove the bias a developer has when testing his own software, and thus increase the probability to find more defects.

Hence, the responsibilities of the software design team are

- Requirements verification
- Design verification
- Code verification
- Module testing
- Module test verification (i.e. verification of the module test w.r.t. the detailed design specification)
- Integration testing

The responsibilities of the software validation team are

- Validation testing

2.3 ECSS Compliance

The RUAG process model is aligned to the ECSS standard, since most of RUAG's customers demand this. As described in section 2.1, RUAG needs to comply with the following reviews:

- *Software Requirements Review (SRR)*
- *Preliminary Design Review (PDR)*
- *Critical Design Review (CDR)*
- *Quality Review (QR)*
- *Acceptance Review (AR)*

The mapping of the phases of the RUAG software development process to the ECSS standard is shown in Figure 4. Thus, the figure shows which activities are performed in between the ECSS reviews. Note that the ECSS not only requires the reviews to be taken out but also requires certain VAs to be performed. RUAG complies to this requirement by having the same VAs as demanded by the ECSS (plus an additional code inspection) and follows the demands imposed on each VA by the ECSS.

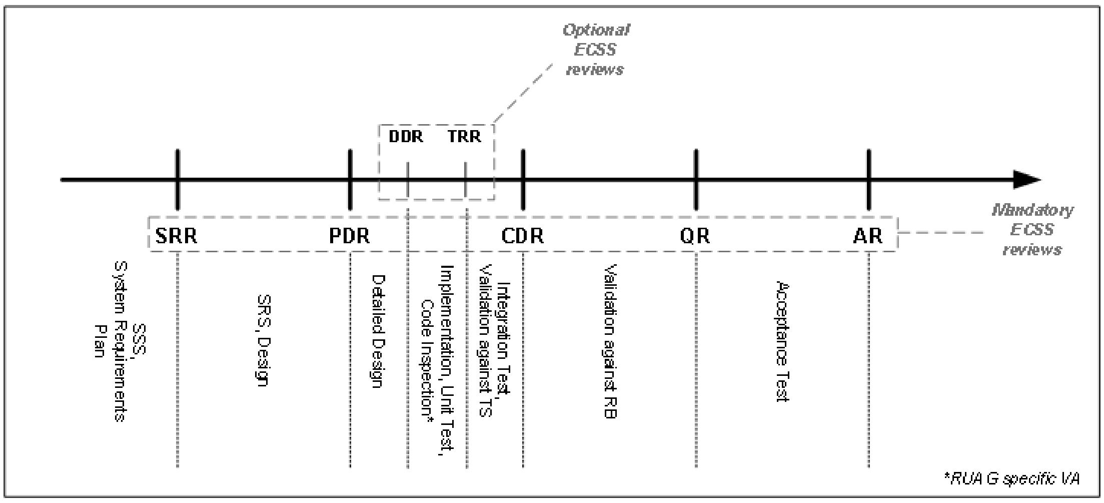


Figure 4: ECSS compliance of the RUAG software development process

The software system specification (SSS) along with further system and project planning work is done by the system team until the SRR.

The more detailed software requirements specification (SRS) and the software detailed design (SDD) are then developed until PDR and reviewed. I.e. the design review DR performed by RUAG can be mapped to the PDR.

This activity follows a detailed design that further specifies the interfaces and describing what each function does and how it is done. The detailed design is completed with the detailed design review (DDR), which is not mandatory in the ECSS standard.

After DDR, the code is implemented, unit tested and code inspected, and is completed with the test-readiness review (TRR), which is also not mandatory in the ECSS standard.

The integration test and validation against the technical specification is finished until the CDR.

Between CDR and QR, the validation against the requirements baseline is performed.

Finally, the last activity in the RUAG development process, the acceptance test, is conducted until the AR.

2.4 Verification & Validation Activities

This section describes the verification & validation activities (VAs) that are carried out at RUAG in more detail. Note that the requirements review is left out, since it was out of scope of this research and was part of another research project.

2.4.1 Design review

In the design review, the software design document (SDD) is reviewed against a checklist by typically 3-4 team members. The faults and problems found are reported in the review document along with a short description.

2.4.2 Unit test

The unit test activity is performed to verify the implemented module against its specification. The test cases are written and executed by the developer. Which test cases have to be performed is documented in a unit test plan.

The test case selection follows a black-box approach. The internal structure is however also considered to analyze problems within the module (white-box approach). Following the black-box approach, the SW is tested without consideration of the internal structure. The selection of the test cases is hereby based on

- Equivalence partitioning
- Boundary-value analysis

Additionally, test cases are selected to test mathematical algorithms within the module (e.g. for synchronizing the time, checksum creation, etc.) should be checked. This is rather a grey-box view of the module. The test cases should take into account:

- Nominal values
- Extreme values
- Accuracy
- Overflow/Underflow
- Division by zero

The white-box test cases are selected based on different coverage criteria, which depend on the project. In a typical project, the source code is tested for statement coverage, branch coverage and condition coverage.

Furthermore, the module is robustness- and stress-tested. Robustness concerns testing invalid input, which is not already covered with the equivalence partitioning. Although stress testing is usually performed on a higher level like e.g. integration test, single modules may also be stress-tested for allocation of resources, filled-up queues, or the like.

A module passes the unit test when all the required test cases have been successfully executed and passed.

Tool support

The Unit Test activity is created and performed by the developer of the module himself, but cross checked by another developer. The main tools that are used in this activity are Splint [SPL07] and Cantata. Splint is a tool developed by the *Inexpensive Program Analysis Group* at the *University of Virginia* and allows the static checking of C programs for security vulnerabilities and coding mistakes. It provides more extensive checks than compilers, such as e.g. unused declarations, type inconsistencies, use before definition, unreachable code, ignored return values, execution paths with no return, likely infinite loops, etc.[UOV03]. Cantata++ is a tool developed to efficiently perform unit and integration testing and is developed by the IPL Information Processing Ltd [CAN09]. It allows the automated execution of white and black box test cases [IPL09].

RUAG does not prescribe the developer when to perform the unit tests. Therefore, a developer might develop and run the tests at the end of implementation, while another might develop and run the tests already during implementation, as can be seen in Figure 5.

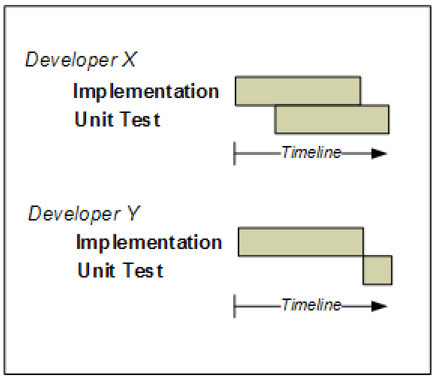


Figure 5: Differences in the use of unit testing

Regression activity

Unit test regression is performed by updating the unit tests to ensure that the black box and white box criteria are met for the modified code and all tests are rerun for the changed module.

2.4.3 Code inspection

In the CI, the code is analyzed using checklists, the logs from the previously described Splint tool and an analysis of different code metrics like LOC, number of return & goto statements, and the maximum nesting level as well as the cyclomatic complexity. The checklist, the Splint logs and the code metrics are stored automatically along with the source code (which is typically 5-10 pages) in a Word file.

Regression activity

The regression tests in this VA may be limited to the part of the source code that is directly or indirectly affected by the change.

2.4.4 Integration test

The integration test aims for dynamically testing the interaction of the modules and is performed on the target hardware platform. The IT is a rather lightweight VA and comprises

- A test of the SW on the HW
- Testing of each calling-called interaction between SW components, i.e. whether all relationships between the modules as indicated in the design occur when running the software
- Testing of each provided functionality of every module at least once

Additionally, timing constrains might also be measured.

The integration test is done for every block defined in the architectural design and is performed automatically using the validation test cases. The calling-called relation is verified by running instrumented software that logs calls during runtime.

Regression activity

The regression test is performed by analyzing if new calling-called relations were introduced. The test cases are then updated to cover these new relations. All tests are then rerun.

2.4.5 Validation against the technical specification

The aim of the validation against the technical specification (TS) is to validate that all requirements of the TS are met. This validation is done on the target HW platform. Additionally, the following tests are performed:

Performance test	Verification that the performance requirements are met
Interface test	Test of how the SW communicates with other components, especially the HW
Robustness test	Simulation/injection of errors
Stress test	Pushing the SW to its limits by putting a high load on the resources
Long-time test	Test the SW for a longer time

Table 3: Additional tests performed during validation

The validation tests are performed automatically if possible. Where automated tests are not feasible (e.g. it is not possible to simulate a specific environmental condition), an analysis of the source code is performed with the result of a justification why the test passed or failed.

Regression activity

The regression tests in this VA may be limited to the part of the source code that is directly or indirectly affected by the change.

2.4.6 Validation against the requirements baseline

This VA is similar to validation against the TS, but with respect to the requirements of the RB. Technically, that makes no major difference for RUAG, so that the same test cases as in validation against TS are used, though they sometimes might be adapted slightly. The only difference is that the documentation is done differently, i.e. the traceability matrix is according to the requirements of the RB. The reason for having a second validation activity even though the test cases are almost the same is that this activity is a legal requirement of the ECSS standard.

Regression activity

The regression tests in this VA may be limited to the part of the source code that is directly or indirectly affected by the change.

2.4.7 Acceptance review

The acceptance review is a validation with respect to the ERD, done by the customer or a third party. Apart from providing support with the test environment, RUAG is usually not involved in this activity.

Table 4 shows a comparison of the different VAs.

VA	Test cases	Check-list	Automatic aspects	Manual aspects	Team involved
Design review		X		Checklist based reading	Design team
Unit test	X		Test cases executed automatically	Test case creation	Design team
Code inspection		X	Inspection sheet created automatically	Checklist based reading	Design team (different person than the developer of the module)
Integration test	X		Test cases executed automatically	Test case creation (Partly test case execution)	Design team
Validation test against technical specification	X		Test cases executed automatically	Test case creation (Partly test case execution)	Validation team
Validation test against requirements baseline	X		Test cases executed automatically	Test case creation (Partly test case execution)	Validation team
Acceptance review	X		Test cases executed automatically	Test case creation (Partly test case execution)	Validation team

Table 4: Comparison of verification & validation activities

2.5 Detailed Development Process

The V-model described in section 2.2 describes the formal software development process of RUAG. In an ideal case, the software development would follow a strict waterfall principle as shown in Figure 6.

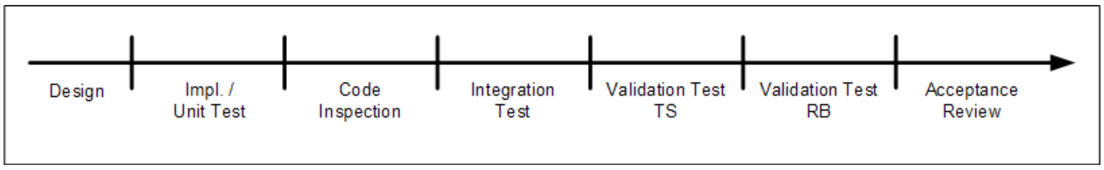


Figure 6: Ideal Process

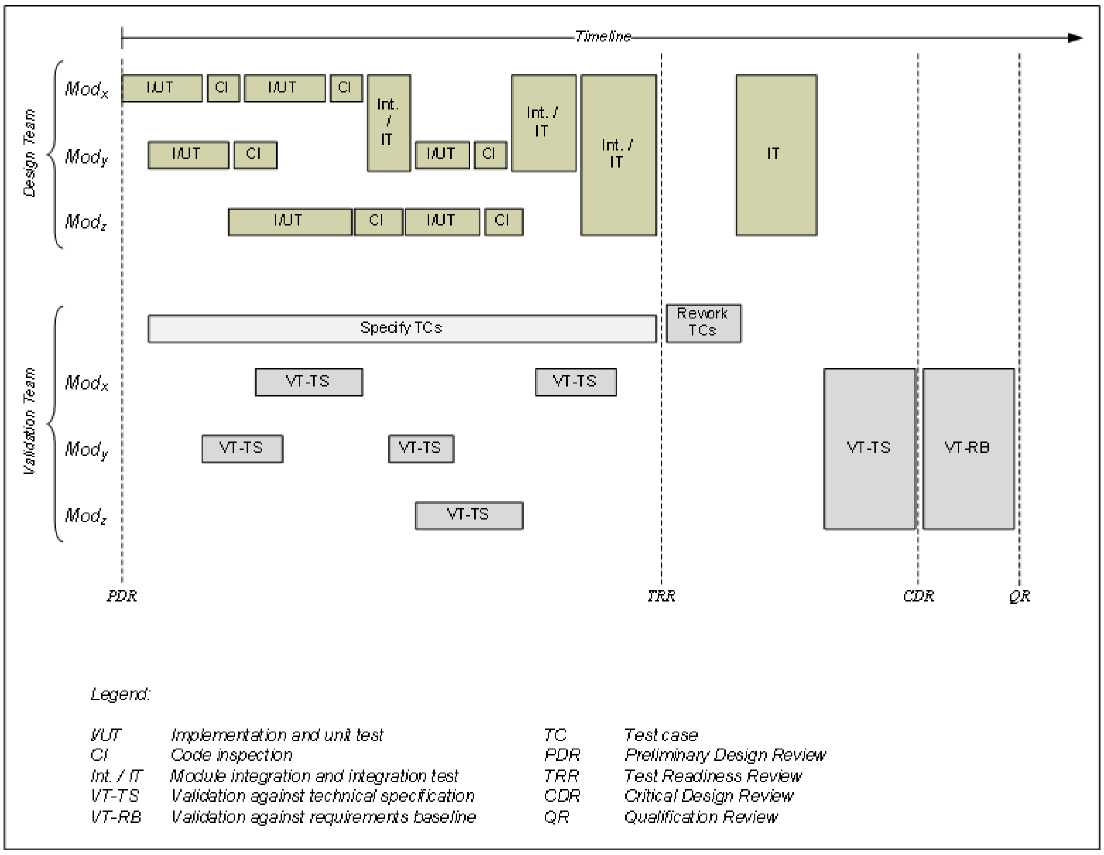
As with any other industry, time-to-market is also a key factor in the aerospace industry, and therefore RUAG develops and tests in parallel to speed up the development and to find faults in the software as early as possible. Hence, although activities are performed sequentially for one module as prescribed in the V-model, several different activities might be ongoing at the same time for different modules. Additionally, the VAs may be executed several times. E.g. if a piece of source code was rejected in a CI, the developer has to modify the code, unit test it again and then, the CI is performed again.

Furthermore RUAG is changing from a traditional waterfall-like to an integration-driven process.

This makes it necessary to take a closer look at the development processes as they are performed in practice for both the former process as it was done until recently and the new integration-driven process as it is done today.

2.5.1 Former process

Figure 7 shows the former process on a fictitious software project that is developed by the design team (upper part) and tested by the validation team (lower part) and contains three modules Mod_x, Mod_y and Mod_z. As shown, the modules are subsequently implemented, unit tested, code inspected and finally integrated. In parallel the validation team starts already to validate each module until the final integrated software is available and the final test campaigns are executed.



Timeline
Design Team
Mod_x
Mod_y
Mod_z
I/UT
CI
Int. / IT
IT
Validation Team
Specify TCs
Rework TCs
VT-TS
VT-RB
PDR
TRR
CDR
QR
Legend:
I/UT Implementation and unit test
CI Code inspection
Int. / IT Module integration and integration test
VT-TS Validation against technical specification
VT-RB Validation against requirements baseline
TC Test case
PDR Preliminary Design Review
TRR Test Readiness Review
CDR Critical Design Review
QR Qualification Review

Figure 7: Detailed view of the former process

A VA for a module is started as soon as the previous activity is finished. This means that for instance when a module is implemented and unit tested, it is directly sent to code inspection. Similarly, the validation team starts already to test in parallel as soon as some modules are finished.

Integration is done after the source code has been unit tested and inspected. The source code can have passed these phases however several times, e.g. when the CI rejected a module. Also, if problems are found during integration, the UT and CI are performed again. The modules are therefore constantly integrated until all modules have been integrated. The end of this phase marks a test-readiness review (TRR). This however differs from project to project. Usually this is an informal meeting where it is decided to pass over to the formal integration test, but can also be externally done by a third party.

In parallel to the software production, the VT-TS test cases are specified, which are also used for the IT. The VT-TS are already executed informally during this time, i.e. before TRR is reached and the formal VT-TS campaign starts. The rationale behind this is to start testing and finding defects as early as possible in the process and better utilize the human resources, since the software validation team performs this activity and not the software development team. Hence, by starting early, the work-intensive test case (TC) development is done early and in parallel so that the TCs are ready to use for the official VT-TS which has to be performed after TRR.

The point in time when this informal execution of the VT-TS test cases is conducted differs from project to project. In one project it was done after a planned time schedule, i.e. not related to the completion of a CI, I/UT or IT. Hence the source code could have been in any state (e.g. not yet finished unit tested) while running the VT-TS. However, the VT-TS test cases for one module usually are not run before the module had been code-inspected at least once.

Usually, a rework of the VT-TS TCs is done after the integration has finished, based on the coverage information obtained from this activity. That means that the TCs at this point in time usually do not completely reflect the coverage demands and therefore have to be modified.

With the same rationale as for the VT-TS, the integration tests are also already performed during integration of the modules. Therefore, several integration test cases have already been executed before the TRR has been reached and the software is officially ready for the integration test campaign. These tests are however performed only informally, and after TRR a formal integration test is performed again.

After the formal IT campaign the VT-TS campaign starts. The difference between these two phases is that the IT is done on modified software which contains additional code for testing purposes such as logging outputs to measure the coverage.

A side-effect of this way of work is that it is not possible to clearly state the start and the end of a VA from a process point of view, since every type of VA might be going on in different modules at different points in time. Hence, when observing the ongoing VAs from the outside, an overlap in VAs can be recognized, as depicted in Figure 8.

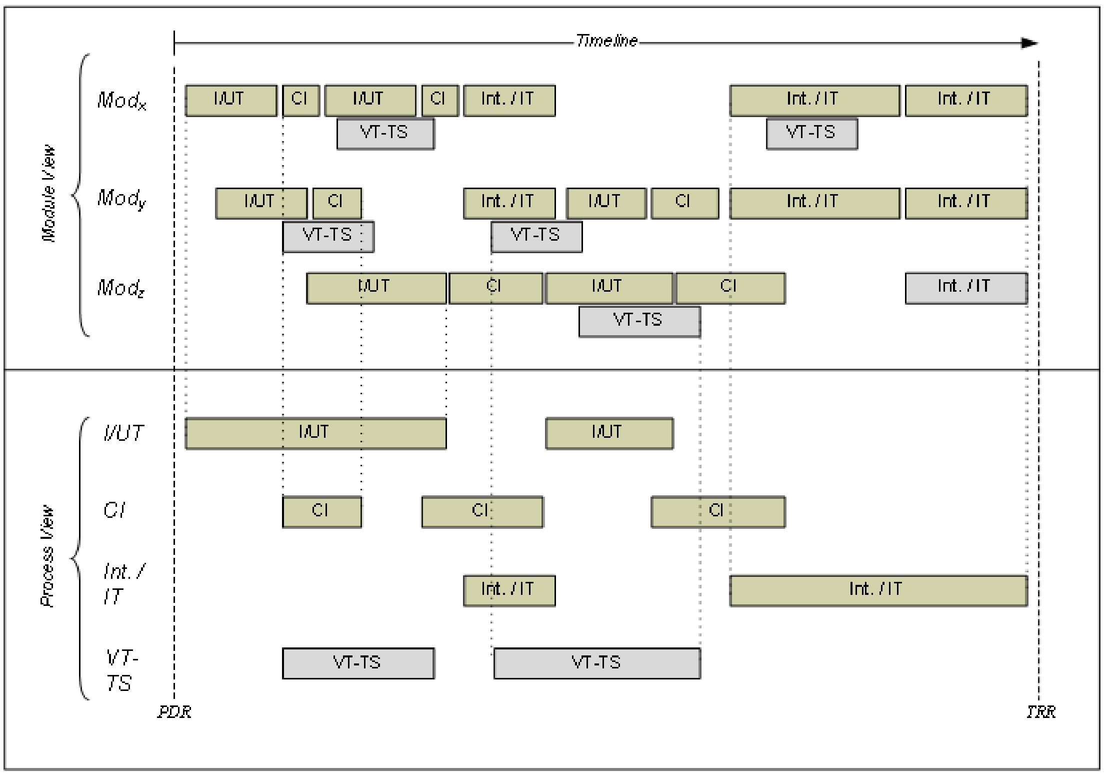


Figure 8: Process view on the former process

The upper part of Figure 8 shows the module view, i.e. the activities that are performed for each module. The lower part shows in contrast the process view, i.e. all activities that are going on at any time during the software development process.

In the module view, the process looks strictly sequential for each module. E.g. CI cannot start while the module is still being implemented and unit tested. Similarly, the integration cannot start before the modules have been code inspected. One exception to this is again the VT-TS activity, since this activity is always done in parallel and independently by the software validation team.

The process view in the lower part shows for each activity when it is performed. In the beginning e.g. a large period of implementation is going on, in which several modules are developed, while in the end, integration becomes a larger part. Thus, when considering all modules at once, then the process is not strictly sequential anymore (lower part of Figure 8). This means, that basically any activity (I/UT, CI, I-IT, VT-TS) might be going on at any point in time in the software production phase. E.g. a CI might be going on while also an I/UT is going on, though for different modules.

2.5.2 Integration-driven process

In the current and future projects, RUAG is doing an integration-driven approach to develop SW as shown in Figure 9. Typically, the SW is developed in three iterations, starting with core functionality (e.g. boot SW) and finishing with more application-level modules. Hence, the main difference to the old process is that the implementation of the modules is not started all at once anymore, but begins with this core functionality. The development of the next component only begins when the previous component is being integrated. As with the old process, problems during integration will trigger regression unit-tests and code-inspections for the modified code.

Another difference to the former process is that the informal execution of the VT-TS test-cases must be done now after the module has been code-inspected at least once.

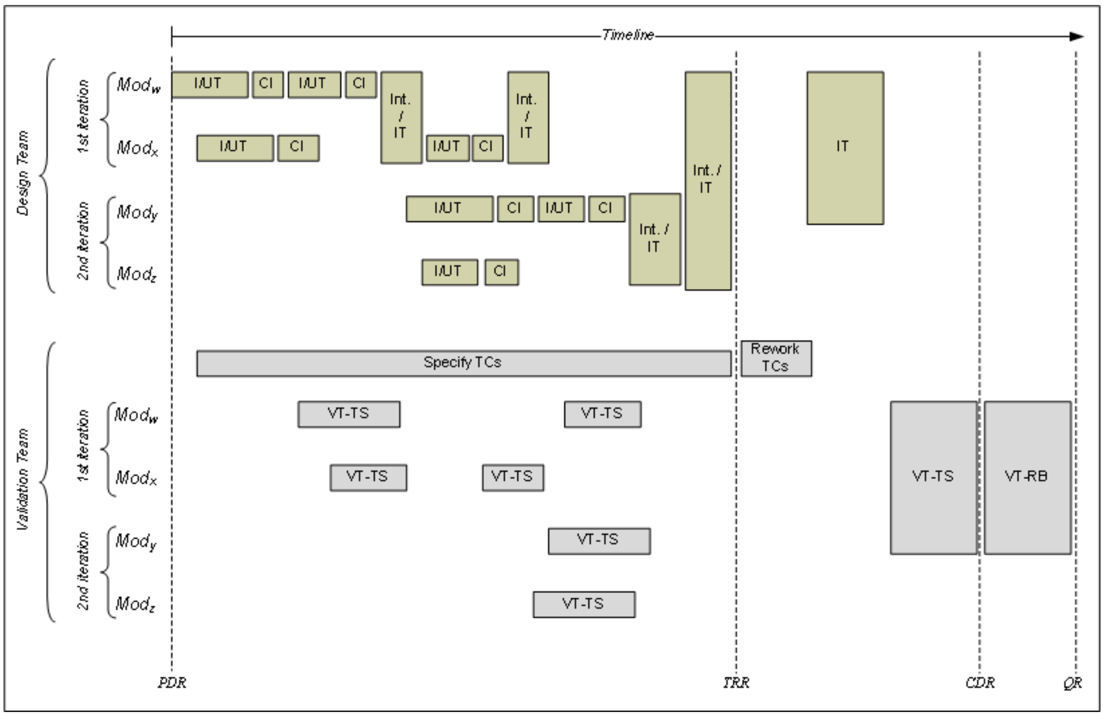


Figure 9: Integration-driven process

Hence, the process view for the integration-driven process looks as in Figure 10. Once again, if only a single module is considered, the activities follow a strictly sequential waterfall-like process. If however all modules are observed at once, any activity might be going on at any point in time, as indicated by the process view in the lower part of Figure 10.

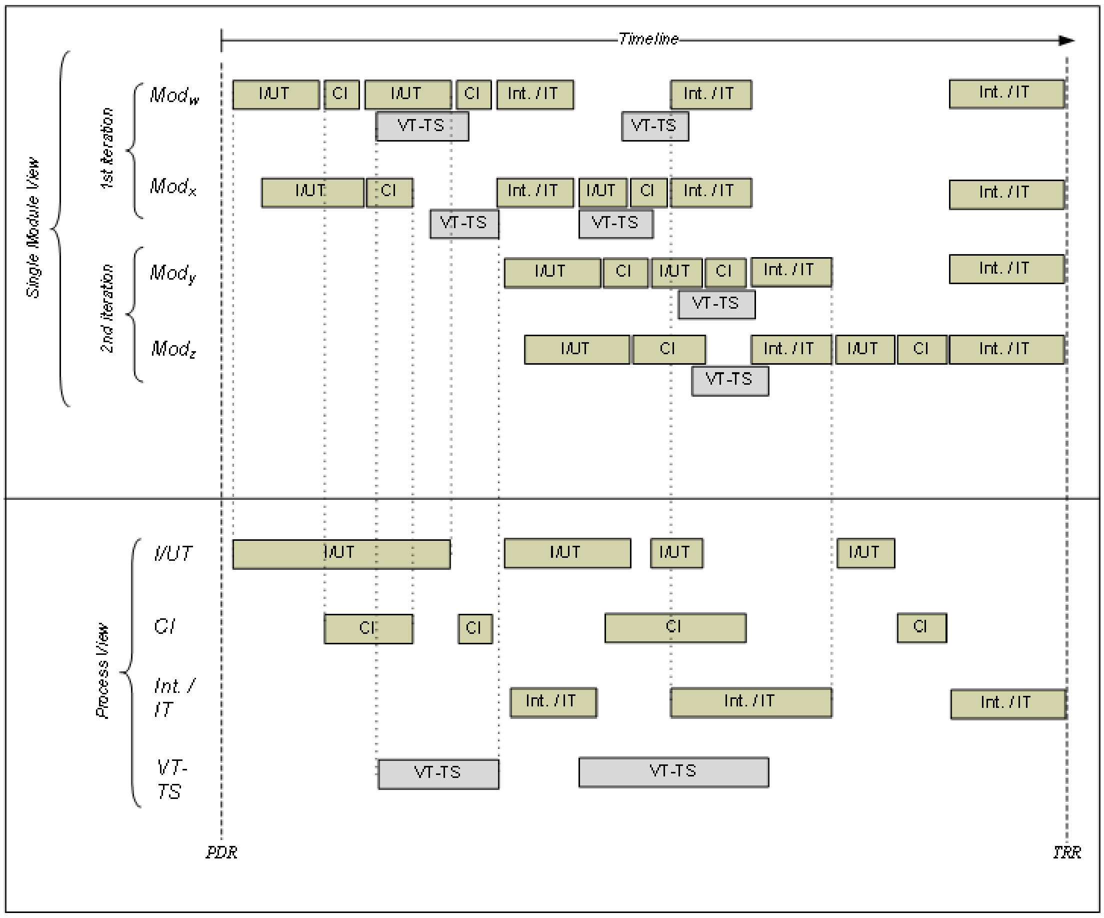


Figure 10: Bird’s eye view on the integration-driven process

2.5.3 Atomic activities

The aforementioned overlapping of the activities and the usage of informal test campaigns in the software development process make it necessary to have a more fine-grained view on the VAs in particular with regard to possible defect data sources.

Table 5 shows the new list of activities, with the newly split activities marked grey. The IT is split into an informal IT (I-IT) and the formal IT which is performed after TRR. Similar is the VT-TS split into an informal VT-TS (I-VT-TS) and the formal VT-TS conducted after TRR and IT.

Abbreviation	**Activity**
DR	Design Review
DDR	Detailed Design Review
UT	Unit Test
CI	Code Inspection
I-IT	Informal Integration Test
(F-)IT	Formal Integration Test
I-VT-TS	Informal Validation Test against the Technical Specification
(F-)VT-TS	Formal Validation Test against the Technical Specification
VT-RB	Validation Test against the Requirements Baseline

AR	Acceptance Review

Table 5: List of atomic activities

Additionally, the following activities might be performed by external third parties:

Abbreviation	**Activity**
X-DR	External Design Review
X-DDR	External Detailed Design Review
X-CI	External Code Inspection
X-VT	External Acceptance Review

Table 6: List of external activities

Regression activities

One might argue that the VAs could also be split into *normal usage* and *regression usage*. Since the test cases are the same in for normal and regression usage, it is not possible that a fault slipped through to a regression usage in UT, IT, VT-TS or VT-RB, that wasn't detected in the normal usage. Therefore a fault slippage to regression usage may only appear in DR, DDR and CI that are reading-based techniques.
One rare occasion where other faults slipped-through faults are found in regression testing is that due to the modification of the source code, the TC needs to be adapted and through this adaption of the TC a slipped-through fault was detected.

Since these two occasions are considered to happen very rarely in reality, it was decided to use the simplified version without the distinction between normal and regression usage to not further complicate the process and the framework. Therefore, this distinction needs to be further investigated in future work.

3 VAMOS FRAMEWORK

3.1 Introduction

This section describes the framework for the management & optimization of verification & validation activities, in the following abbreviated with the acronym VAMOS. In the following, the goals of the framework are recapitulated and the fault terminology is clarified. Section 3.3 gives an overview of the framework and the sections 3.4, 3.5 and 3.6 detail the framework components.

3.1.1 Goals

As outlined in section 1.6.1, the goal of the framework is to create a coherent framework for the management and optimization of the software verification and validation activities. This framework shall allow comparing the effectiveness and efficiency of the VAs in eliminating different types and optimizing the VAs.

The overall goal can be further broken down into the four sub goals with the related questions:

1. Gain more insight into the VAs.
Which VA finds which faults?
Which VA is more efficient / less efficient?
Which VA is more effective / less effective?

2. Find out which faults a VA finds and which faults slip through.
Which VA does not find the faults it is supposed to find?

3. Find out where there is an overlap, i.e. which VAs find the same defects.
Which VAs find the same kinds of faults?
Which VAs should change its focus?

4. Find out how to improve the process.
How do the VAs need to be changed to improve them?

3.1.2 Error, fault, failure and defect

In the literature, there is no clear definition of the words error, fault, defect and failure. The IEEE Standard Glossary of Software Engineering Terminology [IEE610] e.g. defines the words error, defect, fault, mistake and bug basically as synonyms and only notes that in the fault tolerance discipline the terms are used differently. There, a human action, the *mistake*, manifests in the software, the *fault*, which results in a *failure*. The *error* is thereby defined as the amount by which the result is incorrect.
Avižienis et al. define in [AVI00] the term error as the part of the system state which causes a subsequent failure which in turn is defined as that the software does not comply with its service interface. The fault is the actual cause of the error.
Liggesmeyer [LIG08] defines the term failure as an inconsistent behavior during runtime of the software w.r.t. its specification. A defect, fault or bug is the statically existent cause of the failure and the term error is used as the basic cause of a fault.

For simplification purposes, the terms defect, fault, error and bug are used synonymously throughout the thesis for the origin or basic cause of a defect if not stated otherwise.

Since RUAG also attaches great importance on correct and understandable documentation to achieve a better maintainability, problems in the documentation are also considered as defects.

3.2 Related Work

Instead of focusing on a single verification and validation activity, academia [KIT98], [WOO97], [LIT00] and industry [KIK01] show that combining different VAs can be more efficient in finding defects. To overcome the constant market pressure in industry, further investigation is necessary to guide industry on how to select VAs to maximize the defect detection efficiency while minimizing the effort spent. Some initial work that has been done towards the creation of a comparison framework is presented in the following.

In [WOJ07] Wojcicki and Strooper present an iterative selection strategy (ISS) for verification & validation activities. The selection of VAs is hereby refined based on cost-effectiveness data to maximize completeness while minimizing the effort.

In [DAM06], the concept of Fault-Slip-Through (FST) is presented. In this approach, the faults found in each VA are categorized according to which phase they belong to, i.e. in which VA they should have been found. Based on these findings and the effort of the VA, the improvement potential for each activity is calculated.

Since these two approaches, ISS and FST, build up the foundation of this framework, they will be presented in more detail in the subsequent chapters.

Wagner [WAG05] proposes a more analytical approach. The defect detection techniques are thereby compared using an economical metric, namely the return on investment (ROI). The model also considers the effect of combining different defect detection techniques.

A framework for the comparison of testing activities and formal verification is presented in [BRA06]. However, this approach focuses more on studying the synergy and relationship of these two activities and doesn’t give any advice on how this combination can be optimized.

There also exists an IEEE Standard 1061 [IEE1061] related to software metrics and measurement. It defines a methodology for establishing quality requirements and finding, measuring and validating appropriate software product and process metrics.

Apart from these frameworks and models which are developed specifically for the optimization of VAs, there exist also traditional frameworks like QIP [BAS94], Six Sigma [TAY02] or the SEI measurement process [MCA93] that consider improving the software development process in general.

In the following, the general process improvement frameworks and the ISS, FST and the approach by Wagner are explained in more detail, since these frameworks/methods build up the foundation of VAMOS.

3.2.1 Process improvement frameworks

3.2.1.1 Quality Improvement Paradigm (QIP)

In [BAS94] Basili et al. present an iterative process improvement framework called Quality Improvement Paradigm (QIP). QIP consists of six process steps as shown in Figure 11.

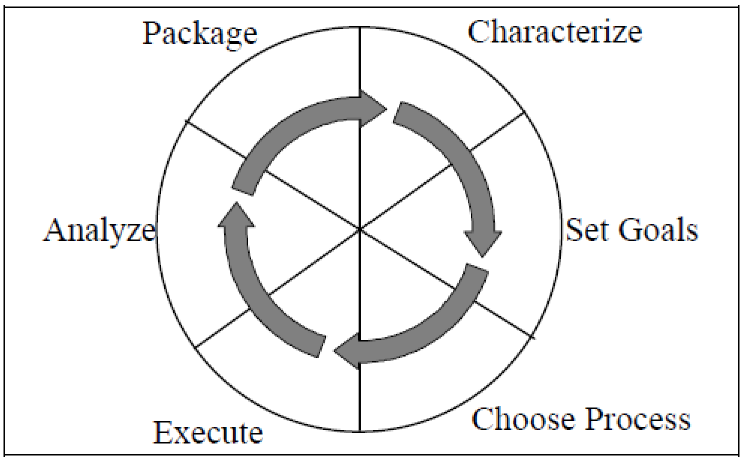


Figure 11: Process Steps of QIP [BAS94]

To understand the QIP process, it is important to know that this process was developed as part of an experience factory to systematically collect the experiences gained in previous projects and learn from that. As such, the QIP process works best for companies that have already conducted several projects and/or are conducting multiple projects at once and thus have a large experience base.
Therefore, the QIP process starts with a characterization step, where the current project is characterized according to existing attributes from the experience base. This is done to find similar projects and thereby set the context for the goals and the process.

After characterizing the process and project quantifiable goals are defined for the improvement of the process.

The next step in QIP is to choose a software development process for the project, like e.g. the waterfall model, based on the previously made characterization.

The project is then developed according to the chosen process in the execute step. The process should thereby reuse existing knowledge from the experience base. Data collection methods must be included into the process in order to judge later-on whether the goals were fulfilled or not.

After the project has been finished, the data is analyzed and any findings such as problems and lessons learned are documented.

In the last step called package, the experience base is updated with the information gained in the project.

QIP is a general framework to improve any kind of software processes [BAS94]. As such, QIP defines what has to be done for continuous software improvement, but not how it is done. It especially relies on GQM (see section 3.2.1.2) to define the measurements that will be used in the project to allow evidence-based process improvement.

3.2.1.2 Goal-Question-Metric (GQM)

In [BAS94-2] Basili et al. define a formal, goal-oriented approach to derive measurements to support the feedback and evaluation of the software process.

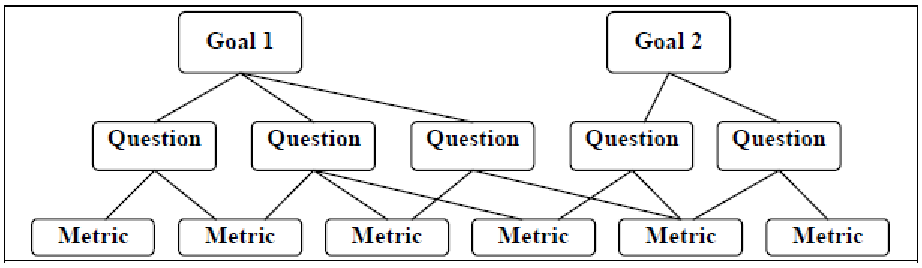


Figure 12: Goal-Question-Metric Approach [BAS94-2]

The methodology consists of three levels as shown in Figure 12, namely *conceptual*, *operational* and *quantitative*.
In the conceptual level, goals are defined for objects of measurements, which can be products, processes or resources. In the context of process improvement, a goal could be e.g. to “improve the timeliness of change request processing from the project manager’s point viewpoint”. Note however, that GQM is not only constrained to process improvement. QGM does not prescribe that the goal has to be an improvement; a valid goal e.g. could also be to understand a process [BAS94-2].
In the operational level, a set of questions is developed which if answered, give information on whether the goal was fulfilled or not.
In the quantitative level, measurements are developed that try to answer a question in a quantitative way. The data obtained from the measurement can be either subjective or objective.

3.2.1.3 Six sigma

In Six Sigma the process is iteratively improved in five phases, namely Define, Measure, Analyze, Improve and Control [TAY02]. Figure 13 shows this iterative DMAIC process of Six Sigma. Since Six Sigma is a general improvement framework, it begins with a define phase, in which the process is analyzed and the problems are discovered. The problem is then quantified through the measurement step. In an analyze step the root-cause of the problem is then discovered and in the improve phase the process is changed to prevent the problem from occurring again. In a final step, the control step, it is assured that the problem does not reappear again and the changes still work.

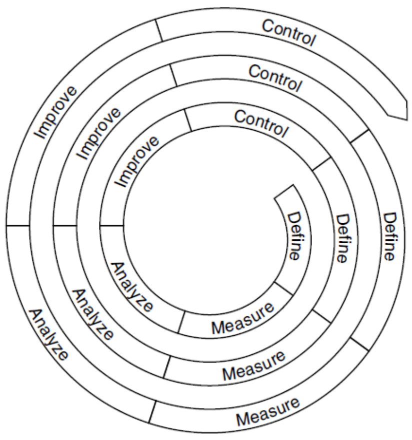


Figure 13: DMAIC process steps of Six Sigma [TAY02]

3.2.1.4 SEI measurement process

In [MCA93] a process for developing a measurement process in a company is defined. Although it is not defined as a process improvement framework but as a framework to derive a measurement process, it's goal is to provide more insight into the processes to make better decisions [MCA93] and thus allowing for a SPI.

It is divided into three parts planning, implementing and improving as illustrated in Figure 14. In the planning phase, the scope is identified, i.e. the problem is defined, and procedures are established for the measurement of the data. In the implementation phase, the data is collected respectively measured. Furthermore the analysis is performed to derive the root cause of the problem. In the last step, changes are derived to improve the process.

Thus, it is similar to Six Sigma and contains the same phases define the problem, measure the data, analyze it and then improve the process. The SEI process skips however the control phase of Six Sigma.

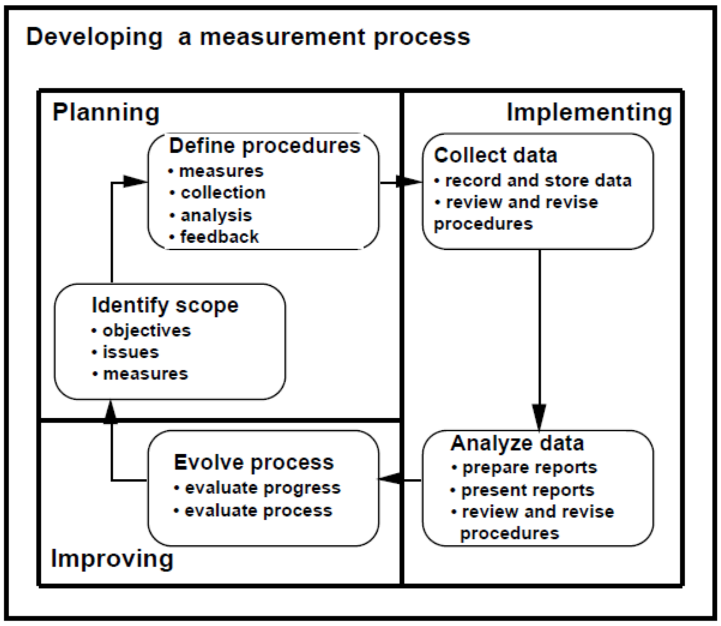


Figure 14: Measurement Process from the Software Engineering Institute [MCA93]

3.2.1.5 Discussion

The processes described above (except for GQM) are based on the well known plan-do-act-check process for business improvement introduced by Deming and Edwards [DEM86]. The basic idea is an iterative process, in which the problem is defined or analyzed, it is further quantified in a measurement phase, then the root-cause is analyzed and finally an improvement is implemented.

Since all the processes aim for general applicability, they lack concrete guidelines to address the problems stated in section 1.4.

3.2.2 Verification & validation optimization frameworks

Apart from these general process improvement frameworks, methods which are more focused on VAs exist, most notably the iterative selection strategy (ISS) developed by

Wojcicki and Strooper [WOJ07] and the concept of fault-slip-through (FST) by Damm et al. [DAM06] which are described in the following.

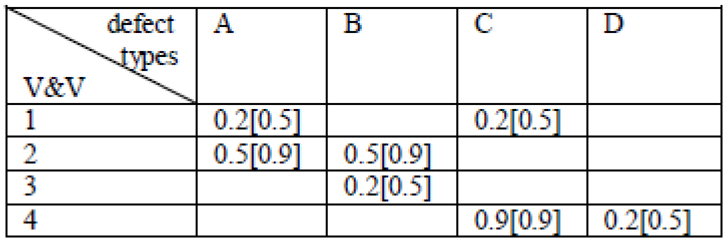
3.2.2.1 Iterative selection strategy (ISS)

A way to select VAs from an existing set of available VAs is presented by Wojcicki and Strooper in [WOJ07]. The approach focuses on minimizing the effort that is spent on the VAs by simultaneously considering the *completeness* of VAs. This means each defect types should be found by at least one VA. The strategy for the selection of the VAs is done iteratively in the sense that it is refined in every cycle with the data obtained from the measurements in the previous cycle(s).

The strategy consists of the following four steps:

1. Gathering of cost-effectiveness information

First of all, the cost and effectiveness data are obtained. That means that for each VA_x and each fault type FT_y, it is determined how effective the VA_x is in finding the defects $FT_{y.}$ and how much effort must be spent to find that fault. If no empirical data are available the data can also be obtained with expert opinion. The effort and effectiveness values should be entered as normalized values into a cost-effectiveness matrix in the form X[Y] as shown in Figure 15, where X indicates the effort and Y the effectiveness. The rows indicate the different VAs and the columns indicate the defect types of interest.

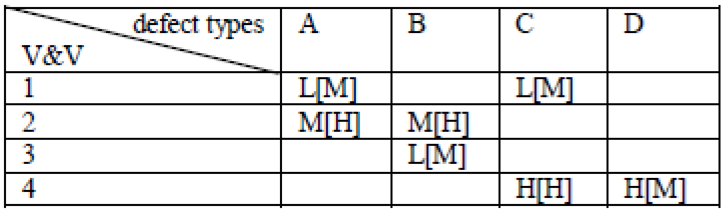


V&V \ defect types	A	B	C	D
1	0.2[0.5]		0.2[0.5]	
2	0.5[0.9]	0.5[0.9]		
3		0.2[0.5]		
4			0.9[0.9]	0.2[0.5]

Figure 15: Effort and effectiveness matrix [WOJ07]

2. Maximizing completeness

In this step the matrix is first transformed into a relative form which shows how the VAs compare to each other, as shown in Figure 16. A simple value range of *High*, *Medium* and *Low* may be used.



V&V \ defect types	A	B	C	D
1	L[M]		L[M]	
2	M[H]	M[H]		
3		L[M]		
4			H[H]	H[M]

Figure 16: Simplified effort and effectiveness matrix [WOJ07]

A selection of VAs is now made that covers all defect types, thus ensuring *completeness*.

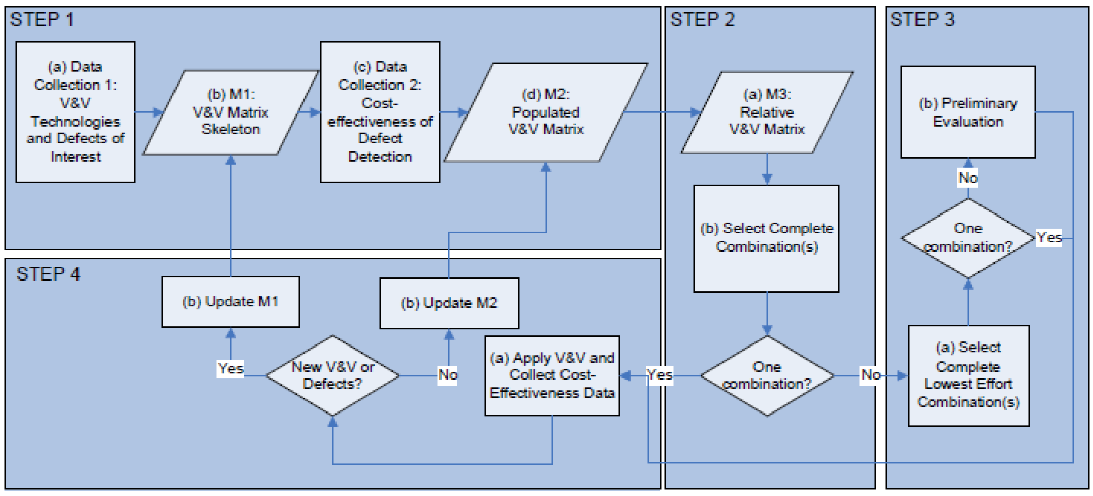
3. Minimizing effort

Several combinations might be possible that cover all defect types, thus in this step the selection with the minimum effort is chosen.

4. Gathering and updating cost-effectiveness data

In the last step of the iterative selection strategy (ISS), the VAs are applied in a project and empirical data is measured to update the cost-effectiveness matrix with more accurate values than the estimates from the first iteration.

The four steps of ISS are shown in Figure 17.



STEP 1
(a) Data Collection 1: V&V Technologies and Defects of Interest
(b) M1: V&V Matrix Skeleton
(c) Data Collection 2: Cost-effectiveness of Defect Detection
(d) M2: Populated V&V Matrix
STEP 2
(a) M3: Relative V&V Matrix
(b) Select Complete Combination(s)
One combination?
STEP 3
(b) Preliminary Evaluation
No
One combination?
Yes
(a) Select Complete Lowest Effort Combination(s)
STEP 4
(b) Update M1
(b) Update M2
Yes
New V&V or Defects?
No
(a) Apply V&V and Collect Cost-Effectiveness Data
Yes
No

Figure 17: Workflow of the iterative selection strategy [WOJ07]

Discussion

The iterative selection strategy provides a systematic way to minimize the effort that is spent on VAs. The problem of this process is that it assumes that the organization has the choice of which VAs it uses. Furthermore does it require the company to have a larger set of VAs to choose from and that a company can change their VAs on a project-by-project basis.

In the Space industry though, the companies are often bound to a certain set of VAs which are prescribed by standards like the ECSS or the customers. Even if the company has not been prescribed which VAs to use, the high dependability and reliability requirements make it necessary to rely on proven defect detection techniques, thus it is not possible for the organization to exchange VAs.

Therefore, the Space industry needs a more fine-grained optimization framework than the ISS, since the choice of the VAs is more conservative than in other industries and cannot be changed that easily. A framework is therefore necessary that focuses more on improving the current VAs and removing overlap on a finer level on the defect types.

3.2.2.2 Faults-slip-through

Damm et al. present in [DAM06] the concept of fault-slip-through (FST) for improving VAs with respect to cost-effectiveness data. In contrast to the traditional opinion, that it is best to find faults as early as possible, Damm et al. reason that it might not always be cost-effective to find them as early as possible. The rationale behind this is, that to find a certain defect directly in the phase where it was introduced might require the introduction of a heavy-weight VA like e.g. conceptual modeling.

Thus, the concept considers the fault-slippage, i.e. the time between where the fault was found and where it is most cost-effective to find, instead of considering the fault-latency, i.e.

the time between where the fault was found and where it was introduced, as shown in Figure 18.

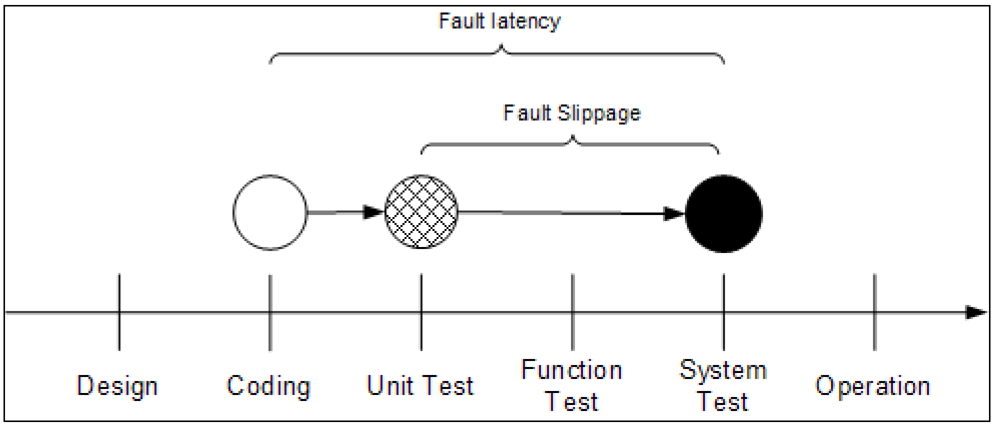


Figure 18: Fault slippage vs. fault latency [DAM06]

Based on the findings, an improvement potential (IP) of each phase is calculated. This value indicates the possible savings if the faults are found earlier and thus is calculated as the difference between the actual fault cost and the fault cost if none of the faults had slipped through.

The improvement potential is determined in three steps:

1. Determine faults that could have been avoided

Before the faults can be classified according to which phase they belong to (PB), a test strategy must be developed that defines what should be tested in which VA and hence, which VA should find which fault. Based on this test strategy, the fault-slip-through is measured and entered into a matrix showing where faults are found (PF) and to which phase they belong (PB), as shown in Figure 19.

PF **PB**	**Design**	**Impl.**	**Function Test**	**System Test**	**Operation**	**Total PB/phase**
Design	1	1	10	5	1	18
Impl.		4	25	18	2	49
Function Test			15	5	4	24
System Test				13	2	15
Operation					0	0
Total Potential PF	1	5	50	41	9	106

Figure 19: Fictitious example of fault-slip-through data [DAM06]

2. Determine average cost

After the categorization the average cost of finding a defect in a VA must be determined for all VAs. The average cost should include the time to find and fix the defect and also the time spent on regression testing. The values might be obtained from a time-reporting system or expert estimation.

Design	Implementation	Function Test	System Test	Operation
1	2	10	25	50

Table 7: Example average fault cost [DAM06]

3. Determine the improvement potential

With the fault-slip-through data and the average cost to find a defect in a certain VA, the improvement potential can be obtained. The IP is calculated for a VA_y and a VA_x and defines the amount of time which can be saved if all faults that slipped through from VA y to VA_x would have been already found in VA_y. Thus, the IP can be calculated for each PB with respect to each PF as shown in Figure 20.

	PF_1	PF_2	PB_{total}
PB_1	IP_{11}	IP_{21}	$PB_{1\ total}$
PB_2	-	IP_{22}	$PB_{2\ total}$
PF_{total}	$PF_{1\ total}$	$PF_{2\ total}$	$(PB/PF)_{total}$

Figure 20: Improvement potential matrix [DAM06]

The formula for calculating the IP for is defined as follows:

$$IP_{xy} = \left(No.faults\ bel.\left(PF_x PB_y\right) \times PF_x AvFC\right) - \left(No.faults\ bel.\left(PF_x PB_y\right) \times PB_x AvFC\right)$$

Thus, it is calculated as the difference between what the fault costs are now (i.e. after slipping through) and what the cost could have been if they had not slipped through and were found in the VA where they were supposed to be found. The calculated sums in the last column therefore define the total IP for a VA and show how many hours can be saved if no fault slipped through the VA at all.

The calculated IPs for the example are shown in Figure 21 and show that the implementation phase has the highest IP.

PF / **PB**	**Design**	**Impl.**	**Function Test**	**System Test**	**Operation**	**Total PB/phase**
Design	0	1 x 2 – 1 x 1 = 1 h	10 x 10 – 10 x 1 = 90h	5 x 25 – 5 x 1 = 120h	1 x 50 – 1 x 1 = 49h	260h
Impl.	___	0	25 x 10 – 25 x 2 = 200h	18 x 25 – 18 x 2 = 414h	2 x 50 – 2 x 2 = 96h	710h
Function Test	___	___	0	5 x 25 – 5 x 10	4 x 50 – 4 x 10	235h

				= 75h	= 160h	
System Test				0	2 x 50 – 2 x 25 = 50h	50h
Operation					0	0h
Total Potential PF		1h	290h	609h	355h	1255h

Figure 21: Improvement potential matrix with sample data [DAM06]

Based on the IP, it can then be decided which VA should be optimized to gain the highest benefit from the improvement.
The concept is further extended in [DAM05] to include fault types. The fault types are however not incorporated into the calculations and are only used to facilitate the analysis.

Discussion

Instead of trying to find the faults in the phase where they were introduced, the concept of FST tries to find the faults where they are most cost-effective to find. Thus it incorporates economic aspects into the improvement considerations and serves therefore as the basis of this framework.

The major drawback of the concept is that it only defines the calculations but does not define how the data can be obtained in measurements and how improvements are derived from the IPs.

Another problem exists when trying to classify to which VA a defect belongs to, i.e. in which VA it should have been found. This requires a well-defined test strategy that clearly states for each defect in which VA it should be found. Since the defects can be very different in their nature, this test strategy would also have to comprise all kinds of defects.

Furthermore in meetings with representatives of RUAG it became apparent that it is difficult to judge when a defect *should* have been found and that it is easier for the developer to say when a defect *could* have been found. Although this looks like a minor difference in wording, the term *should* implicitly requires that the developer knows a test strategy, even if none such exists. Note that this is however still an open issue that has to be solved in future work, as the ultimate goal is to make evidence-based decisions on when a fault should and could have been found to correctly improve the processes.

3.2.2.3 Analytical model of the economics of defect-detection techniques

Wagner proposes in [WAG06] an analytical model of the economics of VAs that is aimed at evaluating and improving VAs. The model uses different fault types and the difficulty of a VA to find these fault types to calculate the costs.

The model consists of the three components direct costs, future costs and revenues. The direct costs are the costs of the VAs which can be directly measured. The future costs are the costs that occur if the defects are not found in one of the applied VAs. Thus, the future costs indicate the costs that occur when the faults slip through to operation. Finally, the revenues indicate the savings that can be achieved if no fault slip through occurs. Based on these three components, a return-on-investment can be calculated that indicates the ratio between the benefits and the costs (or investments) of the VAs.

Discussion

The model presented by Wagner proposes an analytical model of the quality-economics of VAs, which is however designed as an ideal model and not aimed to be practical applicable [WAG05]. As such, the model tries to represent the faults as realistic as possible and furthermore requires probability functions about the effectiveness of the VAs. Wagner considers in particular the different manifestations of a fault in the software artifacts as separate faults. A fault which was introduced in the software design and causes two lines in the source code to be wrong would therefore be considered as three different faults in the Wagner model. Although this is a very fine-grained view on the defects and their corresponding costs, it is too complex to be used in an industry setting.

Furthermore does the Wagner model only consider the revenues gained when no faults slip through to the system operation. Thus, it does not consider the benefits of a reduced fault slippage between the earlier and the later VAs.

The Wagner model builds however the foundation for the cost-considerations of the framework and is therefore described in more detail in the subsequent sections.

3.3 Overview

The basic structure of the VAMOS framework is depicted in Figure 22. The framework follows an iterative process similar to the continuous process improvement frameworks presented in section 3.2.1 and the iterative selection strategy presented in section 3.2.2.1 and consists of four steps measure, analyze, improve and implement. The necessary inputs of the framework are cost and defect data, as well as a fault classification. The iterative process is surrounded by two quality gates, namely the measurement options model (MOM) and the adaptive defect classification (ADC), which the quality of the input measurement processes and the fault classification. A further define phase allows for a parameterization of the framework to adjust it to the company's needs.

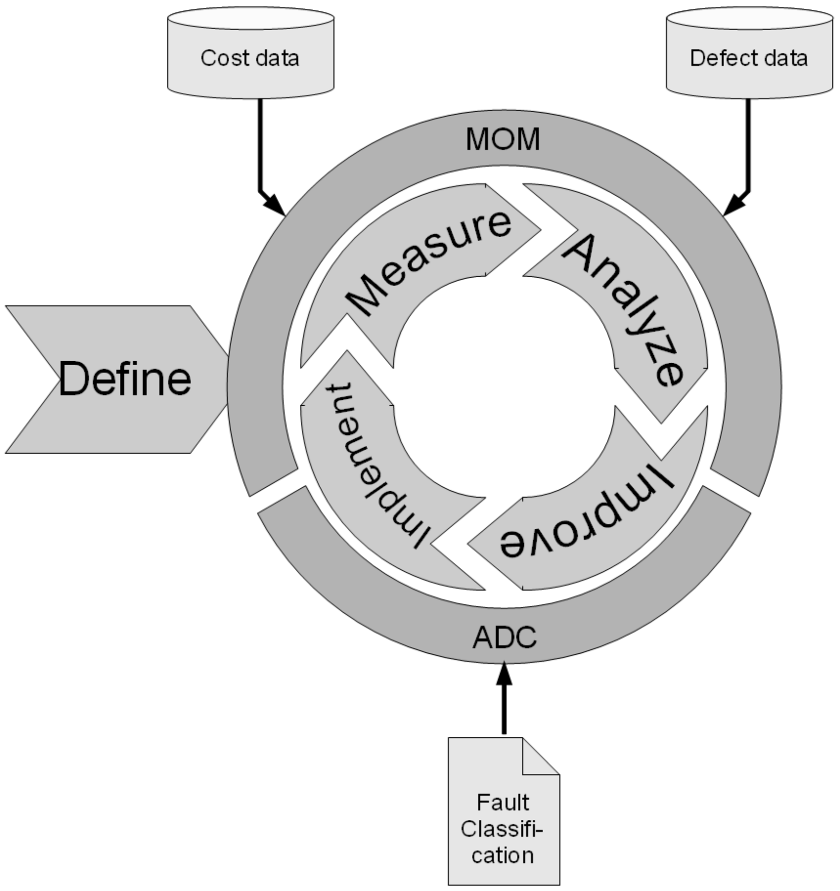


Figure 22: VAMOS framework structure

In the first step, the necessary data is measured. This data is then analyzed to find out the problems in the VAs using the concept of FST. In the improvement phase, adequate changes are developed to address these problems and are then evaluated with respect to the benefit they produce. In the last step, these changes are implemented in the company.

Since this is an iterative cycle, any supposed benefit will be measured in the next iteration and thus, can be directly evaluated.

The process steps are explained and detailed in section 3.6.

As the framework is based on the FST [DAM06] and the ISS [WOJ07], the framework requires certain inputs, which are basically the defect data, i.e. how many faults are found in each VA, and the cost data, i.e. how much does it cost to perform the VA and how much does it cost to fix a defect. Furthermore, a fault classification is needed which is used to classify the defects and facilitate the root-cause analysis.

Section 3.4 gives further information on the inputs.

Typically, companies already have some sort of cost and defect measurement established, e.g. through a work tracking tool and a defect reporting system. In general it is preferred to reuse these existing measurements respectively input data, than to introduce new measurements and ignore already existing data. On the other hand, usually measurements are not always done for every VA of the process and if it is, it’s not always done with the same accuracy. Thus, new measurements may also need to be introduced.

To make sure existing measurements meet the necessary quality to be reused in the framework and to facilitate the development of new measurement processes for VAs that do not yet have one, two *quality gates* surround the framework.

The first one, the measurement options model (MOM), is used to assure the quality of the cost and defect data. The second one, the adaptive defect classification (ADC), on the other hand is used to guarantee the fault classification is usable for the framework or, if it is not, to help develop a new fault classification which is adapted to the company and the domain the company is working in.

The quality gates are described in section 3.5.

In the define phase, the goals and resources for the application of the framework in the company are set. The VAMOS framework allows for a customization to reduce the effort that has to be spent when applying it. A typical tradeoff that has to be made is between the effort spent on measuring data and the accuracy of them. A company may e.g. not have the resources to perform a full application of VAMOS and may have to adjust it to their needs. Thus, VAMOS allows for reducing the measurement effort by providing different tracking options such as sampling the data or estimating them. Furthermore, two alternative options are discussed at the end of this chapter in section 3.7 which allow for a further reduction of the effort.

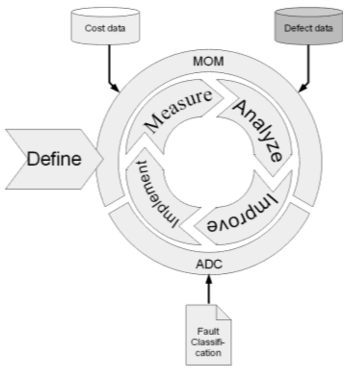
The different options and alternatives therefore act like knobs on an automat that allow for an adjustment of VAMOS to the company’s needs and resources. Hence, the define phase is a crosscut phase that adjusts these knobs in the other phases and the quality gates according to the goals of the company. In particular it is closely connected to the MOM and uses it to setup the VAMOS framework. In the following sections, a *define* subsection is therefore added to the parts of the framework where the framework allows for a parameterization to adjust it to the company’s goals and needs.

Since a reduction of the effort spent on the framework also results in a reduction of the accuracy of the data and therefore a lower reliability of the validity of the analysis and the change proposals, it is necessary to trade-off between a low effort and a high accuracy. Hence, the company applying the framework should answer the question whether the framework should be used to accurately quantify the problems and possible savings, or whether the framework should only serve as a first indicator to reveal the biggest problems.

3.4 Inputs

As described in the previous section, VAMOS requires defect and cost data as inputs which are described in the following.

3.4.1 Defect/effectiveness data

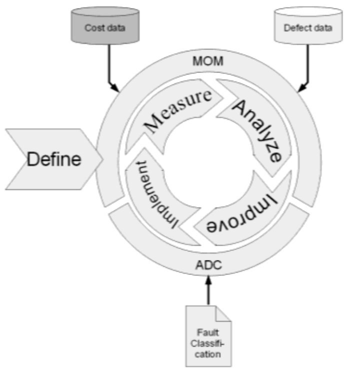


The effectiveness data describes how many defects are found in a VA, hence, indicating how *effective* the VA is in finding defects. In the following, the terms effectiveness data and defect data are used as synonyms.

The defect respectively effectiveness data need to be collected for each VA of the process. Furthermore, it is not sufficient to only track how many defects are found in each VA. As shown in Figure 18, the FST requires that each defect is classified according to the phase where it was found as well as the phase where it could have been found. Additionally, it is necessary to classify each defect according to a fault classification.

To simplify the future analysis, the defect data should also contain information like e.g. the file name and version, to trace the defect back to the source code.

3.4.2 Cost data



For the further considerations of the framework, it is necessary to gain a more thorough understanding of the costs that occur when applying a VA and the simplifications and assumptions that were made in this framework. In [WAG06] Wagner proposes an analytical model for the economics of verification & validation activities, which serves as the basis for the further cost considerations within the framework.

The Wagner model formulas to calculate the cost of the applications of VAs in relation to the number of faults found. The main aspect of the model is the comparison of what the cost would have been if they were found by the different VAs with the cost if the faults were found in operation.

Fault propagation in the Wagner model

The Wagner model considers the propagated faults separately as shown in Figure 23.

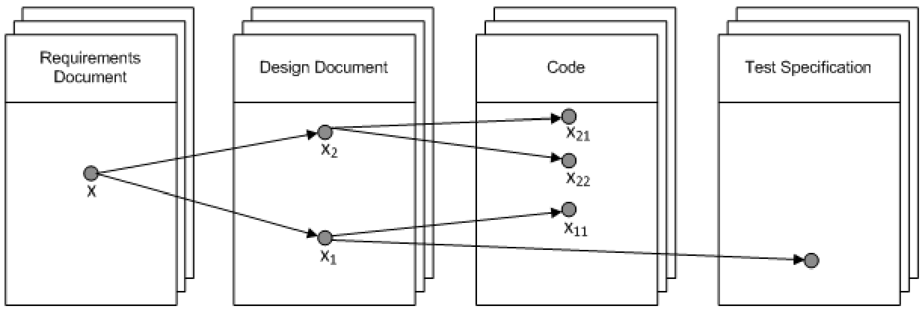


Figure 23: Fault propagation in the Wagner model [WAG06]

That means, that e.g. a fault x which was inserted in requirements and was not found in the corresponding requirements review might result in two faults x_1 and x_2 in the design document. Furthermore, if the design review also fails to find these two defects, they might propagate further to the coding phase and result in three faults x_{11} , x_{21} and x_{22}, whereas fault

x_{11} is the fault resulting from the case that x_1 was not found in the design review and x_{21} and x_{22} the faults because x_2 was not found. Hence, Wagner's model considers each manifestation of the fault in a software artifact as a different fault. Wagner calls the faults that cause further faults in later artifacts predecessors R_i. In the example, the fault x would therefore be the predecessor R of the fault x_1; the fault x_1 the predecessor R_1 of the fault x_{11} and so forth.

Simplification

This detailed differentiation of the defects is however not very useful in the context of fault-slip through, since we want to know which defect could have been found earlier. Therefore, a fault found that slipped-through is considered as the same as its predecessor, but which was found later. Furthermore, several defects that originate from the same predecessor will be considered as the same defect, which however affects different parts of the artifact.

The rationale behind this is, that when a fault is discovered, it is typically necessary to find the origin of the defect to be able to understand and fix it. Since the ECSS prescribes also rigourous tracing of the documents, it is further required to verify all modules that are affected by the changes. Thus, it is considered that whenever the origin of a defect is found, all other, subsequent defects of the origin defect are also found and fixed. E.g. if in the example the defect x_{21} was found, the origin x_2 respectively x is discovered and because of the module tracing, the subsequent defects x_{22}, x_1 and x_{11} are also found and fixed.
Note, that this is just a simplification that was considered necessary and might not always hold true in reality.

To summarize the simplifications, if a fault slipped through the requirements document as shown in Figure 24 and causes the design document to be wrong in two parts, it is considered that it is the same defect, but with two new manifestations in the design document. If it further slips through to the code and causes three spots in the source code to be corrected, it is still considered as the same fault which has now slipped through from the requirements to the implementation.

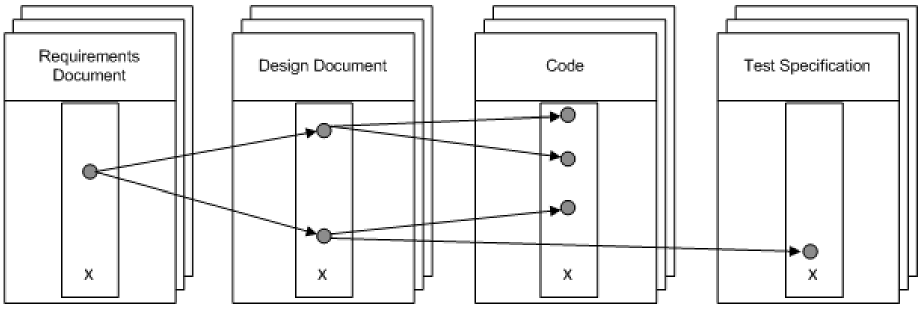


Figure 24: Simplified fault propagation

Costs of VA application

Figure 25 illustrates the cost of the application of a VA_A in the Wagner model. Basically, the cost consist of fixed setup costs u_A and the cost to execute the activity $e_A(t)$. Furthermore, the cost of removing $v_A(t)$ a defect needs to be added, if the defect was found.

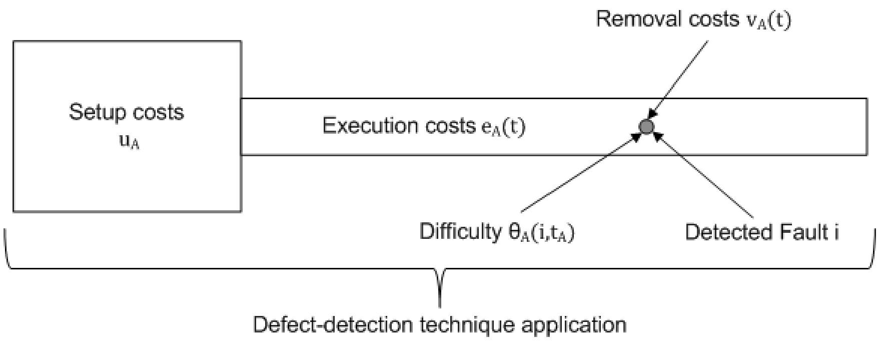


Figure 25: Breakdown of the cost in the Wagner model [WAG06]

The major drawback of this approach is that the model does not describe the removal cost in detail and thus, gives no information on how the removal costs $v_A(i)$ are calculated and how regression activities are incorporated into the formula.

Detailed cost structure

To better understand these removal cost in the context of space applications, a more detailed analysis of the removal cost is necessary which is shown in Figure 26. The removal cost $v_A(i)$ are split up into the costs to find a defect $f_A(i)$, the costs to fix it $x_A(i)$ and in the cost of the regression activities $r_A(i)$.

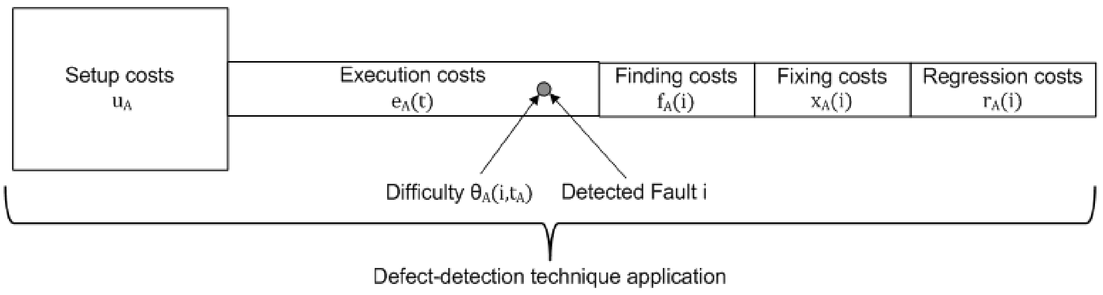


Figure 26: Detailed breakdown of the costs

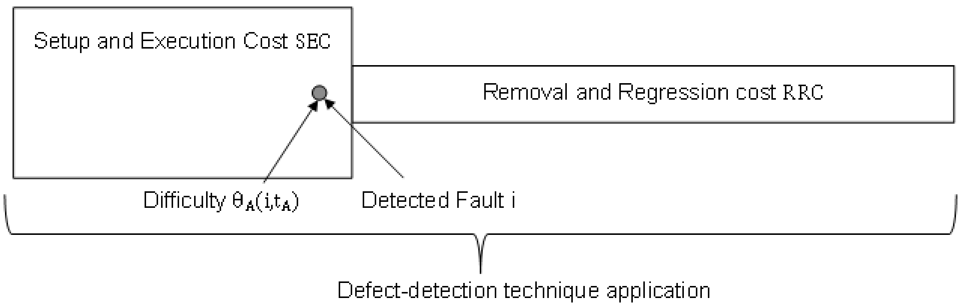
In the Wagner model, the setup costs denote mainly the cost to configure the test environment, while the execution cost are considered with the test execution, like e.g. performing reviews or running the test cases. Wagner considers that the execution time of the VA can be varied and therefore considers the execution cost as variable, depending on the length of the application, i.e. the time.

Simplification

In the context of space applications and the ECSS standard, this separation into setup- and execution cost is however not really suitable. E.g. the running of the test cases does usually not require a lot of time. Test cases that run several hours to days are performed automatically during the night or at the weekends. Furthermore, since the ECSS standard or the customers demand certain coverage criteria, spending a variable amount of time on a VA is not really possible. Therefore and for simplification reasons in the cost measurement process, the setup- and execution costs were combined to one cost (SEC).

Additionally, the finding & fixing costs were combined as well together with the regression cost as the removal & regression cost (RRC). Although these costs may vary depending on the fault (e.g. a certain defect might be easy to find, but difficult to fix, while another one is very hard to detect, but easy to fix), it was deemed necessary for simplification reasons. Furthermore are the finding & fixing costs in practice usually measured in conjunction.

Figure 27 shows the new cost structure of a VA with the newly introduced SEC and the RRC. The SEC is the initial cost of a VA (e.g. specification and execution of test cases, performing the code inspection, etc.) which is the combination of the setup costs and the execution costs from the Wagner model. The RRC are the costs that are caused by the defects (i.e. finding the defects, fixing them, performing regression activities). The RRC are the combination of the finding & fixing costs and the regression costs from the refined Wagner model.


Setup and Execution Cost SEC
Removal and Regression cost RRC
Difficulty θA(i,tA)
Detected Fault i
Defect-detection technique application
Figure 27: Simplified cost

Note that the SEC is measured for the whole VA, while the RRC is measured w.r.t. a single defect. Thus, the SEC build the fixed cost w.r.t. to the number of faults found within the VA, while the RRC build the variable cost. The SEC does therefore not change, regardless how many faults are found in the VA. It changes however with the size of the project. The variable cost on the other hand change with the amount of defects found in a VA. Since the RRC is measured w.r.t. one defect, the total RRC is the sum of the number of faults found times the RRC.

Figure 28 shows this correlation between the number of faults found within an activity and the SEC & the total RRC. As can be seen, the total RRC increases with the numbers of faults found in the VA, since more time needs to be spent on finding and fixing the defects and performing regression testing. The SEC however stays constant, since the VA has to be set up and performed anyway, no matter how many defects are found. E.g. test cases have to be written which adds to the SEC, even if no faults are found at all.

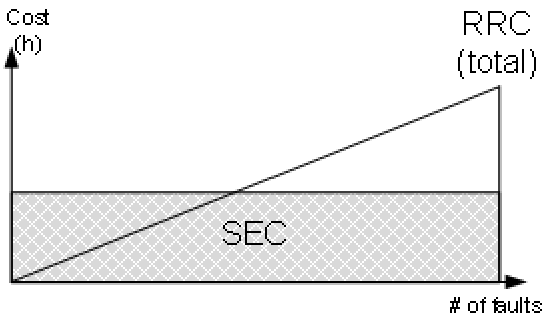


Figure 28: Correlation of SEC and RRC with the number of faults

Figure 29 illustrates the two different cost types on an example of a VA. Initial tasks like specifying test cases and running them are considered as SEC. All other costs are incurred by the faults are counted as RRC. Thus, the finding & fixing of the defects and the performing of the necessary regression activities (such as UT, CI, etc.) are RRC.

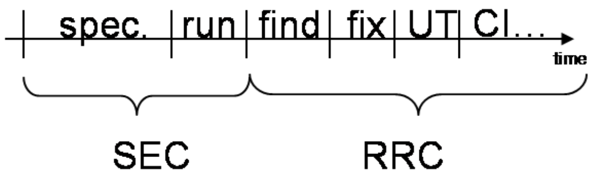


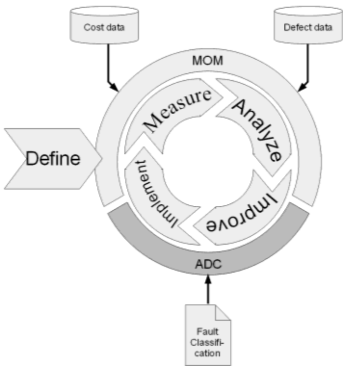
Figure 29: SEC and RRC in relation to VA steps

3.5 Quality gates

The quality gates define a mechanism to derive efficient cost & defect measurements and a domain-specific fault classification. Furthermore, it defines ways to evaluate existing measurement processes and fault classifications with respect to their suitability for the VAMOS framework. Hence, the quality gates serve as a filter that an existing measurement or data source has to pass to be allowed for usage in VAMOS.

There exist two quality gates, the adaptive defect classification (ADC) and the measurement options model (MOM). The ADC is for evaluating respectively deriving a domain- and company-specific fault classification. The second quality gate, MOM, serves for evaluating respectively deriving the effectiveness and cost measurement processes.

3.5.1 Adaptive defect classification (ADC)



The fault classification plays a crucial part in all further analysis of the VAs at RUAG, since an inaccurate classification scheme may lead to data that does not reflect the process accordingly and incorrect conclusions might be drawn when trying to improve the process. Hence, an accurate classification scheme was needed. Freimut et al. [FRE05] point out what generally makes a fault classification *good*:

- *Orthogonality*
  Each fault fits only into one category
- *Completeness*
  Each fault can be classified into at least one category
- *Consistency*
  Different analysts classify a defect into the same category

Furthermore, the fault classification should only contain few classes which should be applicable to every phase and product, otherwise it will be difficult for the developer to select the correct one [CHI92].

If a fault classification is already available at the company and the classification fulfills the required quality characteristics, it can be used for the VAMOS framework. Otherwise, a new fault classification has to be developed. In the following, a process is described to derive a fault domain-specific fault classification.

3.5.1.1 Related work and development methodology

The topic of fault classifications has been extensively discussed in literature and is still topic of further discussion. As shown in the next section, several fault classification schemes exist, which are either meant to be generally applicable or domain-specific. Furthermore, processes have been developed to derive fault classifications.

General classifications

The two most widely used fault classifications [WAG08] are the *orthogonal defect classification* (ODC) developed by IBM [CHI92] and the *defect, origins, types and modes* developed by HP [GRA92].

In ODC, the defects are classified according to different sets of attributes, where the most important ones are triggers and defect types. The triggers define the circumstances under which a defect emerges and the defect type refer to the type of defect according to how it was fixed.

In the classification from Hewlett-Packard, the defects are also classified according different attributes (called dimensions in the Hewlett-Packard scheme). The three dimensions are origin, type and mode, whereas the origin is the activity in which the defect could have been prevented for the first time, the type is a further refinement of the origin into the area that is responsible for the defect and the mode indicates why the defect occurs.

There is also an IEEE standard [IEE1044] which also defines a set of attributes to classify the defects. The IEEE standard is however not frequently used [WAG08].

Apart from these classification schemes that aim for general applicability, there also exist classifications tailored to specific domains. Of special interest with respect to the space domain and the foundations of the framework is also the fault classification developed by the NASA [LUT03] as well as the fault classification used in the FST by Ericsson [DAM05]. Both classifications are based on the previously mentioned ODC

Table 8 shows a comparison of the defect type attribute aforementioned defect classification schemes. Note, that although the ODC, Hewlett-Packard and the IEEE scheme define different attributes to classify a defect, only the defect type is considered and compared. Other defect attributes are e.g. severity of the defect or the artifact where the defect occurred (design, source code, documentation, etc.). In the context of the defect analysis, the defect type is however the most important attribute since it defines the cause of the defect.

	ODC v5.11	**IEEE**	**HP Origins, Types and Modes**	**Faults-slip-through**
Domain	**General**	**General**	**General**	**Telecom**
Attribute	**Defect Type**	**Type**	**Type**	**-**
	Assignment/Init	Data handling	Data handling	
	Checking			Robustness
	Algorithm/Method	Computation	Computation	Logic
		Logic	Logic	
		Data (Exc. Handling)	Error Checking	
	Function/Class/Object		Functional Description	
			Internal module design	
	Timing/Serialization	Timing problem	Interprocess communication	Concurrency
	Interface/OO-Messages		Module/Obj interface	Internal interface
			SW Interface	
		Interoperability	HW Interface	External interface
	Relationship	Subroutine mismatch		
			User interface	Human interface
		Performance		Performance
		Standard conformance	Standards	
		Document quality problem		

Table 8: Comparison of defect classification schemes

Existing development processes

In [FRE05] a process for creating and validating a customized fault classification in an industrial setting is proposed. This process comprises three phases. In the initial phase the defect classification is created, consisting of two attributes *detection* and *injection*, i.e. where it was found and where it was injected. This is called *defect flow model*. Since the injection data are usually not available, Freimut et al. propose to estimate them through expert opinion. Based on this model, the *QA-Strategy* is developed, indicating which VA can find which defect easily or difficultly.
In the second phase the *correction type* is defined, answering the question of what was fixed when correcting the defect. This is validated through interviews. The scheme is further refined and validated in a third phase which is an application in a pilot project.

Nakamura et al. present in [NAK06] an approach to derive a defect classification from the change history in a source code versioning system. This approach is aimed at organizations that do not track defect data or where the defect data is not in a useful form to derive a defect classification scheme. The idea of this process is that the defects which exist in the intermediate versions of the source code, i.e. the versions between the initial commit and the final commit, are analyzed. Based on these low-level findings, a high-level defect classification is abstracted. This is done in an iterative fashion. First an initial set of source code that the analyst is already familiar with is examined. After these, code with big changes is selected to get further evidence. The analyst then tries to group them, and if a good grouping could be found, the classification scheme is abstracted from that.

3.5.1.2 Fault classification development process

Although the aforementioned fault classifications strive for general applicability, they did not suit the needs of RUAG without having to modify them. The development processes described in the previous section on the other hand have the disadvantages that they either rely only on a single software artifact [NAK06] or they require a lot of team involvement [FRE05]. Thus, a new process for creating a classification scheme was developed that focused on analyzing all VAs and software artifacts and requires a lower developer involvement. The process allows for a fault classification regarding the defect type attribute, as this is the most important attribute in the context of the VAMOS framework. A generalization of this process that considers other defect attributes is currently under development with the name adaptive defect classification [FEL09-2].

The process is depicted in Figure 30. It starts with the development of an initial fault classification which is taken from literature, since it is usually better to rely on an existing, proven classification scheme and adapt it to the specific domain and organization [FRE05] than to develop a completely new scheme. The classification is then iteratively refined by applying it on sample defect data in every VA. In a last step, the classification undergoes a quality review with the involved stakeholders (i.e. developers, testers, project managers) where final adjustments are made.

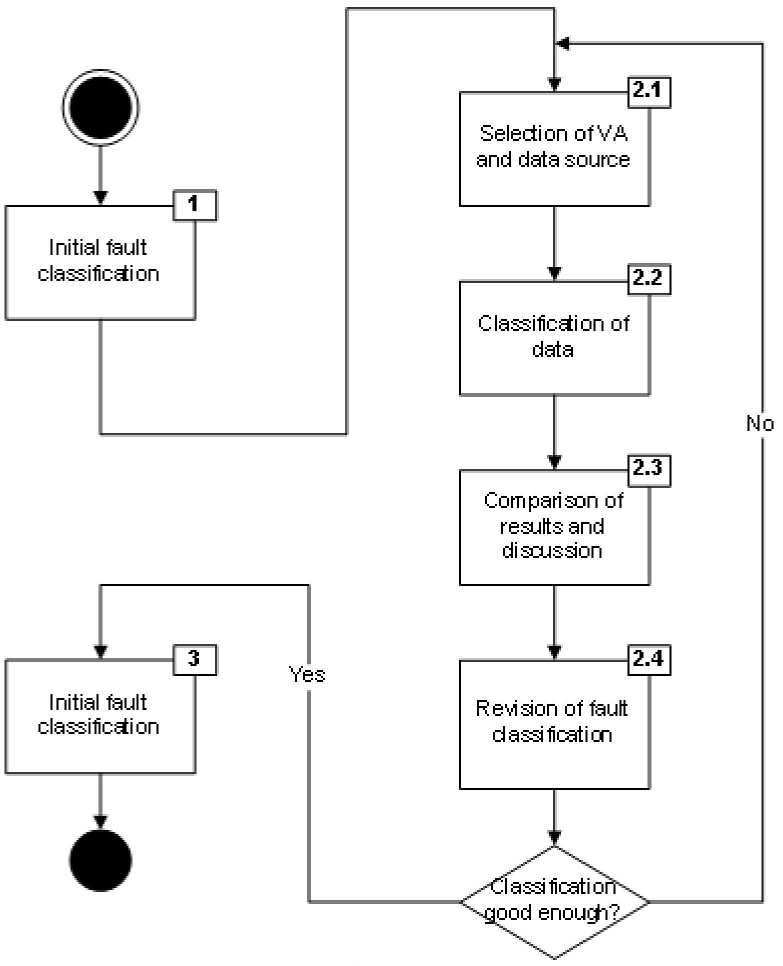

1
Initial fault classification
2.1
Selection of VA and data source
2.2
Classification of data
2.3
Comparison of results and discussion
2.4
Revision of fault classification
No
Yes
3
Initial fault classification
Classification good enough?

Figure 30: Fault classification development process

Step 1

In the first step, a proven fault classification from literature has to be taken which is similar to the domain of the company. The classification can already be adapted to the company based on previous experiences and assumptions and does not have to be very exact, since it is refined in the following steps anyway.

Step 2.1 – 2.4

The classification is now iteratively refined until the classification is assumed to be good enough. At first, the data source for the VA that should be analyzed is selected. This can be defect reports, commit comments, inspection sheets, etc. If possible, data from different projects should be chosen. The data (or a randomly selected subset of the data) is then classified using the current fault classification (i.e. the initial fault classification in the beginning) independently from each other by two or three persons. The results are then compared and any discrepancies are discussed and the fault classification is revised.

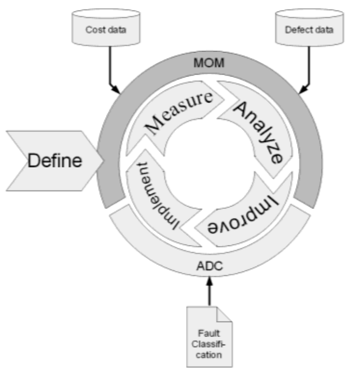
These steps are performed until the quality of the classification is found to be good enough. A way of validating the consistency of a defect classification is presented by El Emam and Wieczorek [EMA98]. Thereby, a factor is calculated that represents the agreement of two analysts to classify the collected defects into the same group. The value of this factor can than indicate whether there exist an *inadequate*, a *good* or an *excellent* agreement and therefore allow to judge objectively when the classification is good enough and the quality review can begin.

Step 3

In the last step, the classification is reviewed in a workshop or through questionnaires by the members from the team.

3.5.2 Measurement options model (MOM)

The role of the measurement options model is to guide industry to find a way to measure the effectiveness and cost data and to ensure that the measurements are as simple as possible and require a minimal effort.



The data can usually be measured in different ways. The cost data can e.g. be measured using a simple spreadsheet, in which the developer periodically enters the time spent on each task. Another option could be to install a tool on the developer's computer that tracks how much time was spent on each tool. A third option could be a web-based work tracking tool where the developer also enters periodically his work information. Hence, different options, in the following called *measurement options*, exist to measure the required data.

The measurement options model is therefore a means to get the best possible option in terms of effort and accuracy to measure the cost or effectiveness data.

3.5.2.1 Quality characteristics

The measurement options vary among each other in the three quality characteristics *variable effort*, *fixed effort* and *accuracy* as described in Table 9.

Variable effort	The effort (in working-time) that a developer has to spend when logging the effectiveness, RRC or SEC information. E.g. it might take the developer 30 minutes to enter a newly found defect into a defect reporting system.
Fixed effort	An initial investment (in working-time or a monetary value) that has to be spent upon introducing the measurement. E.g. a defect reporting tool needs to be purchased and installed and the developers have to be trained in it.
Accuracy	Different measurements record the data with different accuracy. E.g. if a measurement requires the defect reporting to be done after the project or test campaign, the accuracy of this measurement will be lower, since it is more difficult to classify a defect *a posteriori*. This is due to the fact that the person classifying the defect has to reread the code and understand the origin of the defect and the way it was fixed. This makes it very likely that mistakes creep into the classification. The accuracy of a measurement option will on the other hand be high if the defect is classified directly after it is found and fixed. The accuracy furthermore depends on the way the information is recorded and the sampling size.

Table 9: Quality attributes of effectiveness measurement options

Since the options differ among each other and even within the VA they are used in, a general recommendation of which measurement should be used in each VA would be desirable. This is however not useful for the following reasons:

Differences in the VAs

Different companies use different VAs. E.g. one company does perform a code inspection, while another doesn't, but does e.g. model checking instead. Furthermore do companies use VAs differently. A company might e.g. perform its unit test using a test tool, while another one does it manually. A recommendation to automatically log the defects by the tool with the developer only having to classify it afterwards would therefore not be useful for the company that does its unit testing manually. A general list would therefore need to comprise all possible VAs in all possible variants which is not realistic.

Differences in the intensions and possibilities

Furthermore, companies have different intensions and possibilities. A company might e.g. want to spend more variable effort on a measurement to obtain the maximum accuracy.

Measurement options need to be clearly specified

To understand how a measurement should be introduced and used, it is necessary to clearly specify the measurement. This however makes it very specific and adapted to a company, and thus contradicts with the idea of a generally applicable measurement recommendation.

Due to these reasons, no concrete recommendations are given on how to measure the defect data in each VA. But instead a process is described in the following which allows the company to obtain a suitable cost and defect measurement for each VA.

It is further divided into the defect measurement options (MOM-DEMO) for the defect and RRC data, and the setup and execution cost measurement options model (MOM-SECMO) for the SEC data.

3.5.2.2 Defect measurement options (MOM-DEMO)

Figure 31 shows the measurement options model for defect data (MOM-DEMO). The process begins with the question whether a measurement for the VA under consideration already exists or not. If a measurement process already exists, the left path with the steps 1.1 – 1.3 has to be followed to assure that the existing measurement fulfills the quality requirements. Otherwise the right path with the steps 2.1 – 2.6 has to be followed to create a new measurement process for this particular VA.

Steps 1.1 and 1.2

If a measurement process already exists for the VA, the next step is to check whether the quality is acceptable for the use in the VAMOS framework. This means that the following questions need to be answered:

- Is the defect logged with information to trace it back to the software artifact where it appears, e.g. the part of the source code or design or the paragraph in the documentation?
- Is the defect classified with the information where it could have been found and where it actually was found?

- Is the defect classified using the fault classification?

If these questions can be answered and the measurement contains all the necessary information, the next step is to assure that the information is also accurate. This step can be left out if it is decided to keep the existing measurement process in order to reduce effort at the expense of accuracy.

If both, the quality and the accuracy are acceptable, the process proceeds with the analysis of the RRC measurement (step 1.3); otherwise the measurement process has to be newly created by starting with step 2.1.

Step 1.3

Additionally to the defect data, the RRC have to be measured. If the RRC are not yet measured along with the defect reporting, an RRC measurement has to be developed and introduced. This step is as simple as define possible ways of measuring the RRC cost besides the existing defect measurement and select the best option in terms of effort or accuracy.

Step 2.1

At first, for each VA that does not have an adequate measurement process, a list of possible ways of measuring the defects has to be developed (*measurement options catalog*). This can be done in a brainstorming meeting with developers involved in the VAs.
As described in the previous section, a general recommendation on how to measure is not useful. In the following, a list of with typical measurement options is given as, which serves as an initial catalog and starting point and can be evolved by the company. Further ideas on the measurement options can be found in the case study (see section 4.3.2.1).

Introduction of a bug tracker

The effectiveness and cost data can be measured with a bug tracker such as the commonly used Bugzilla [BUG09].

Reuse of the existing infrastructure

If a defect reporting system already exists in the company it may be possible to reuse and modify it in such a way that it supports the tracking of the necessary information.

Modify the versioning system

If a source code versioning system is used it might be adapted to incorporate the measurement whenever a developer commits a bug fix. This might be done e.g. by providing a comment template where the developer has to fill in the required information.

Spreadsheet

A very simple way of measuring defect data is to provide a spreadsheet in which the developer enters the defect information.

Modify the test tool

If a test tool is used, it may be possible to modify it to automatically log any defect found. The defect classification and cost measurement might be done by providing a popup when a test case fails or by filling out a template in the stored test logs.

Step 2.2a and 2.2b

When enough options to measure the defect data have been gathered they need to be attributed with their variable effort, their fixed effort as well as the accuracy. This can be done by listing all measurement options in a matrix with columns for variable effort, fixed effort and accuracy.

Define - Sampling

Since some activities can find quite a large amount of faults, it is important to consider whether all defect data should be collected and classified or not. If e.g. an activity which is performed early-on finds thousands of (minor) faults, it might neither be desired nor necessary to measure all the defects. Thus, a possibility to reduce the measurement effort is to introduce sampling and hence to only report a certain percentage of the defects (e.g. every 10th defect).

The usage of sampling can however have a significant impact on the fixed effort (e.g. if the introduction of sampling requires a modification of the test tool). Therefore it should be decided in advance on how the measurement is performed and the fixed effort for each option should be rated with this in mind.

Step 2.3

As in Step 1.3, the RRC data has to be measured along with the defect data. Therefore, for each possible measurement option from the list, it has to be decided on how the RRC can be measured using this options. E.g. an existing bug tracker might already include a field to specify the costs that were spent on finding & fixing the defect. Therefore this option does not require a separate measurement of the RRC, but is rather *integrated* already. If the RRC cost cannot be logged directly with the option, e.g. if the bug tracker does not support a cost field, then another way to measure the RRC has to be defined.

The measurement of the RRC adds to the effort of the defect measurement option. Considering the example of the bug tracker that does not support a cost field, a way of measuring the RRC could be to store the cost data separately in a spread sheet. This however would require the developer to switch between applications when classifying the defect and thus, increase variable effort (since it takes longer) and decrease accuracy (since the developer might forget to track the RRC in the spreadsheet). Hence, for each measurement option in the matrix, the way of measuring the RRC needs to be defined and the attributes of the option needs to be updated.

Define – RRC measurement

The RRC measurement per fault type may be left out if the alternative framework options are followed, which are described in the end of this chapter (see section 3.7).

Step 2.4 – 2.6

When the matrix has been updated accordingly it can be sorted to retrieve the best measurement option. The sorting is done first by variable effort, then second by the accuracy and third by the fixed effort. The rationale behind this order is that the most important aspect is that the variable effort, i.e. the effort to measure each defect, is rather low to minimize the time the developer has to spend on defect reporting and to avoid too much distraction from his actual work. Furthermore the accuracy is rated higher than the fixed effort, since it is more important to have accurate data than to have a low initial investment. This is because inaccurate data which is used for analysis might lead to incorrect conclusions and thus, wrong improvements of the processes. Although the fixed effort of a measurement option is important as well, it is usually considered the case that a process improvement program lasts for several years and thus the investment will amortize over time.

Depending on the goals of the company, this order can be changed. A company might e.g. want to invest more variable effort for achieving a higher accuracy.

When the best measurement option has been chosen, the process is finished and the measurement option needs to be implemented in the company and the employees need to be trained in how to measure the defects.

If sampling was chosen in the step 2.2, it is also important to find out the right sampling rate to get enough data for performing a meaningful analysis (step 2.5). This can be done by implementing the measurement process in a pilot project and evaluate if enough defects are reported.

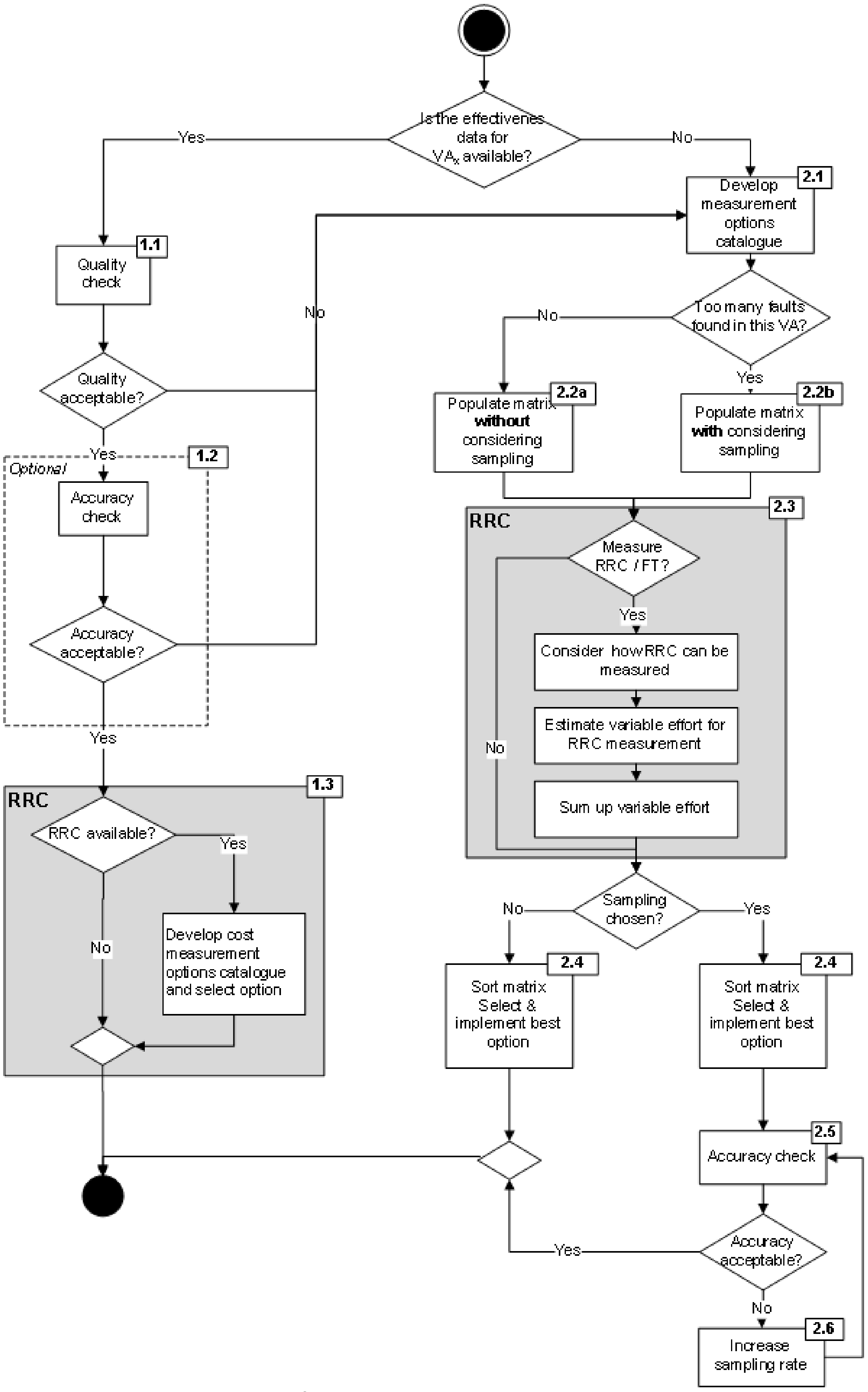

Is the effectiveness data for VA_x available?
Yes
No
2.1
Develop measurement options catalogue
1.1
Quality check
No
Too many faults found in this VA?
No
Yes
Quality acceptable?
2.2a
Populate matrix **without** considering sampling
2.2b
Populate matrix **with** considering sampling
Yes
Optional
1.2
Accuracy check
RRC
2.3
Measure RRC / FT?
Yes
Consider how RRC can be measured
Accuracy acceptable?
Estimate variable effort for RRC measurement
No
Sum up variable effort
Yes
RRC
1.3
RRC available?
Yes
No
Develop cost measurement options catalogue and select option
Sampling chosen?
No
Yes
2.4
Sort matrix Select & implement best option
2.4
Sort matrix Select & implement best option
2.5
Accuracy check
Accuracy acceptable?
Yes
No
2.6
Increase sampling rate

Figure 31: Defect Measurement Options (MOM-DEMO)

3.5.2.3 Setup and execution cost measurement options model (SECMO)

The measurements for the SEC are developed using a similar model to MOM-DEMO and is illustrated in Figure 32. Depending on whether a measurement process already exists for the VA under consideration the quality of the measurement is analyzed (steps 1.1 and 1.2 in the left path) or a new measurement is developed (steps 2.1 – 2.4 in the right path).

Steps 1.1 and 1.2

The quality requirements for the SEC measurement are that the SEC is measured without including any RRC data, i.e. no finding & fixing cost and no regression cost. Thus, the following questions need to be answered:

- Is only the SEC data measured, i.e. only initial cost spent on setting up the test environment and performing the activity?
- Are costs spent on performing regression activity excluded from the SEC?

If the quality requirements are met, an accuracy analysis might follow to assure that the cost data is measured accurately enough. This means that no big variances exist in the measurements and it is clear for every person involved what is in- and what excluded in the SEC measurement.

If the quality requirements are fulfilled and the necessary accuracy is given, the measurement can be used for VAMOS. Otherwise the steps 2.1 – 2.4 have to be followed to create the measurement.

Step 2.1

As with MOM-DEMO, a measurement options catalog has to be developed containing different ways of measuring the SEC data for the specific VA. Again, giving a general list is not useful and therefore only some typical options are described.

Reuse of the existing infrastructure

Usually companies have some means to perform a work tracking. This may be used to track the SEC data.

Spreadsheet

A very simple way of measuring the SEC data is to provide a spreadsheet in which the developer has to fill in the time spent on doing setup & execution activities.

Step 2.2

The measurement options have then to be rated according to their variable effort, their fixed effort and their accuracy.

Step 2.4

In the end the rated measurement options are sorted by variable effort, then by accuracy and last by fixed effort with the same rationale as in MOM-DEMO. The process then finishes by selecting the best option and introducing the measurement in the company.

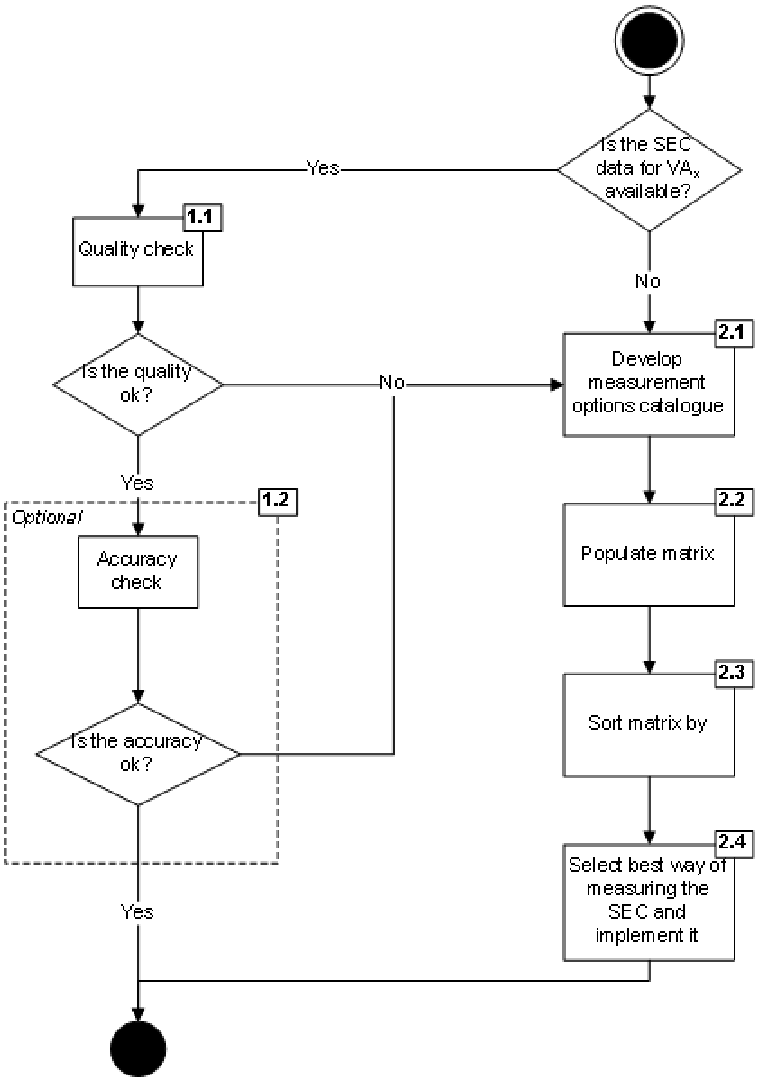
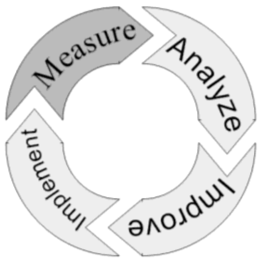


Figure 32: Setup & Execution Cost Measurement Options (MOM-SECMO)

3.6 Process Steps

3.6.1 Measure



In the first step of the framework application, the measurements are carried out in order to obtain the data to be used in the analysis. As described in section 3.4, the necessary inputs for the framework are the effectiveness data, the setup & execution cost as well as the removal & regression cost.

The measurement process is divided into the measurement of the SEC data and the measurement of the effectiveness data, which also includes the collecting of the RRC data.

3.6.1.1 SEC measurement

Figure 33 shows the basic concept of the setup & execution cost measurement process.

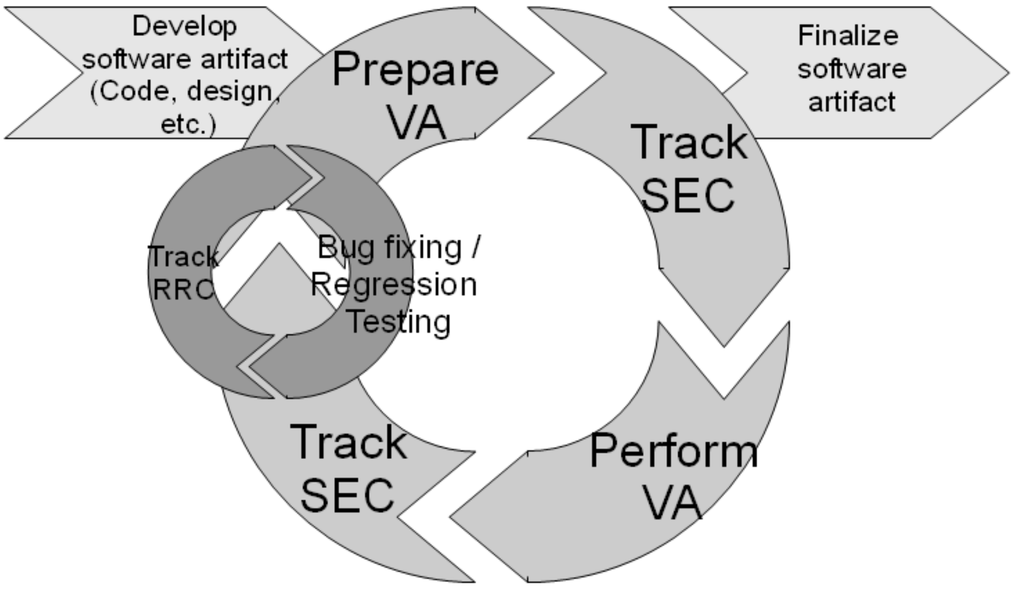

Develop software artifact (Code, design, etc.)
Prepare VA
Track SEC
Finalize software artifact
Track RRC
Bug fixing / Regression Testing
Perform VA
Track SEC

Figure 33: SEC measurement process

After the code has been developed, it is going to be tested within several VAs before it can be finalized and delivered to the customer.

The basic procedure of each VA is:

1. Preparation
2. Execution
3. Finding & fixing defects
4. Perform regression testing

Before the VA can be performed, usually some preparative tasks at the beginning of each VA have to be carried out. Since these tasks count as setup & execution cost, they have to be tracked during or after completion of the preparation. Example tasks are:

- Preparation of testing hardware
- Writing test cases and test scripts
- Preparation of code inspection templates

After the preparation has been finished and all related costs have been tracked, the next step is the execution. This means basically any task that have to be carried out to fulfill the VA. Example tasks are:

- Executing the test cases
- Perform the (first) design review
- Perform the (first) code inspection

This execution cost also count as SEC and therefore have to be tracked as well.

After this first initial execution of the VA the finding & fixing of the defects begins. The bug fixing goes along with performing regression activities. The cost of the defect fixing and the accompanying regression activities are measured in the effectiveness measurement process described in section 3.6.1.2.

All further time spent on the VA is considered as removal & regression cost, and thus, the SEC measurement continues with the next VA.

Define – Tracking options

The standard way of tracking the SEC information is that every person performing a preparation or VA execution-task records the time spent on it. Measuring all the cost may however be too much effort for the company and therefore two options exist to reduce this effort.

Sampling

Instead of measuring all the cost, only a sample, i.e. a subset, can be used.

Estimation

A further reduction of measurement effort can be achieved by simply doing an expert estimation on the SEC data.

3.6.1.2 Effectiveness and RRC measurement

The measurement process for the effectiveness and RRC data is outlined in Figure 34.

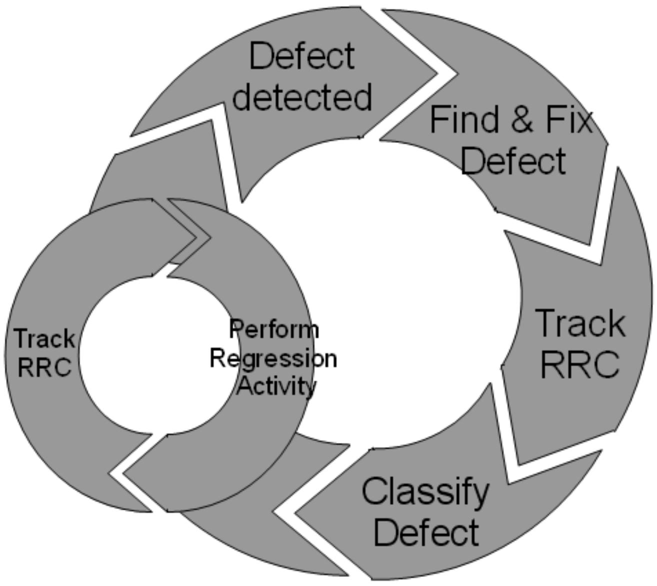


Figure 34: Effectiveness and RRC measurement process

The process begins with the detection of a defect through a failed test case, a code inspection annotation or the like. The defect is then found & fixed and the hours spent have to be attributed to the RRC.

The defect then needs to be classified according to

- its fault type
- the phase where it was found
- the phase where it could have been found

Additional information has to be collected to allow tracing the defect to the source code or design.

Depending on the measurement option for the VA (see section 3.5.2.2), the classification of the defect may also be done *a posteriori*, i.e. after the VA or the project has been completed. Furthermore, the measurement option may support automatic tracking of the phase found and the file information.

After the defect has been fixed and classified, the regression activities have to be carried out. The time spent on the regression activities have also to be added to the RRC data.

Define – Tracking options

The removal & regression cost for a defect have to be tracked by every person involved. That means, that the person finding the bug, the person fixing the bug, as well as any person involved in performing regression testing need to attribute the time they spent on it to the RRC.

Similar to the SEC measurement options exist to reduce the measurement effort.

Sampling

If the measurement option does not yet prescribe a sampling for the defect measurement, the RRC tracking may be sampled separately. Thus, although every defect is measured, the RRC is not tracked for every defect but only for a subset of them.

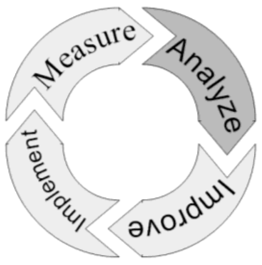
Estimation

Instead of that every person that is involved in the defect fixing & regression testing adds the time to the RRC; another option is to estimate the RRC that the defect will cause. Thus, the person fixing the bug will estimate the RRC of all persons involved.

Although this reduces the accuracy of the measurement, it may be useful for simple faults where estimation may be performed with a high accuracy and an extensive RRC tracking would cause unnecessary effort. Estimation may also be more appropriate for later activities when several people are involved in the defect fixing and regression testing, since there, the tracking becomes more and more complicated and therefore less accurate.

A further problem with measurement occurs when bug fixing or regression testing is performed in batches, which means that several defects are fixed at once or the regression testing is done for multiple defects. Hence, an accurate measurement may be not possible or very difficult, and therefore estimating the RRC may be an alternative.

3.6.2 Analyze



The goal of the analysis phase is to use the measured data to find out in which VA problems exist and what the problems are. In particular, the following questions shall be answered:

1. Which activity does not find the faults it is supposed to find?
2. In which activity could a change pay off?
3. Do different VAs find the same type of faults, i.e. is there an overlap in the VAs?

The analysis phase consists of two parts, the calculation of the improvement potential (IP) and the calculation of an overlap factor (OF). The IP allows to answer the first two questions while the OF answers the third.

3.6.2.1 Data preparation

Effectiveness data

At first, the measured defect data has to be grouped according to the VA they slipped through and the VA they were found in. This can be done in a table where the VAs where the defects are found are on the x-axis and the VAs where the defects could have been found are on the y-axis. Table 10 shows such a table filled with fictitious data. E.g. in the CI column, one can see that 5 faults that were found in CI could have already been found in RR, 10 faults already in DR and 2 faults in UT. Furthermore CI found 40 faults that could have only been found in CI.

	DR	UT	CI	I-IT	I-VT-TS	F-IT	F-VT-TS	VT-RB	AR
RR	4	0	5	3	1	0	0	0	0
DR	20	1	10	30	5	4	1	1	2
UT		3	2	0	0	0	0	0	0
CI			40	5	2	1	0	0	4
I-IT				47	20	5	10	5	3
I-VT-TS					5	0	5	2	0
Σ	24	4	57	85	33	10	16	8	9

Table 10: Example of effectiveness data

Note that in contrast to the Damm model, this table has to be created for each fault type and filled only with the defect data of the corresponding fault type.

Cost data

Similar to the effectiveness data, the cost data has to be grouped according to the VA.
Table 11 shows an example of the measured cost data grouped to the VA. Note that the RRC data has to be also split according to the fault type, which is not shown in this example.

	RR	DR	UT	CI	I-IT	I-VT-TS	F-IT	F-VT-TS	VT-RB	AR	Operation
SEC	10	30	50	70	40	80	10	20	10	0	0
RRC	5	1	2	2	5	8	40	60	80	100	150

Table 11: Example of cost data

3.6.2.2 Improvement potential

The improvement potential (IP) indicates the possible savings due to a removed fault slip through. The IP is calculated for a VA_y with respect to the faults that slipped through to the later VA_x. Thus, the IP indicates how much hours could be saved if the faults where found already in VA_y. According to Damm [DAM06], the IP is calculated as follows:

$$IP_t(x,y) = \left(numberOfFaults_t(x,y) \times RRC_t(x)\right) - \left(numberOfFaults_t(x,y) \times RRC_t(y)\right)$$

The IP is the difference of what the faults cost if they are found in VA_x and what the fault cost could have been if they were already found in VA_y. Hence, a positive IP value denotes the time that can be saved if the faults are found in VA_y.

To gain the total IP of a VA_y, i.e. the time that could be saved if no faults slipped through VA_y, the sum of all IPs has to be taken:

$$IP_t(y) = \sum_{j=y+1}^{n} IP_t(j,y)$$

In contrast to the model of Damm, the IP is here calculated separately for each fault type FT_t to overcome the drawback indicated in section 3.2.2.2.

The total IP has to be calculated for each VA_y and each fault type FT_t. Finally, the highest IP indicates the VA with respect to the fault type where an improvement will return the biggest benefit.

Table 12 shows the improvement potentials for the different VAs calculated from the example data of the previous section. The table contains the IPs of each phase with respect to the phase where the faults could have been found (on the y-axis). The total improvement potentials in the rightmost column are the calculated sums of the IP of each row.

IP	**DR**	**UT**	**CI**	**I-IT**	**I-VT-TS**	**IT**	**VT-TS**	**VT-RB**	**AR**	**IP/VA**
RR	-16	0	-15	0	3	0	0	0	0	-28
DR		1	10	120	35	156	59	79	198	658
UT			0	0	0	0	0	0	0	0
CI				15	12	38	0	0	392	457
I-IT					60	175	550	375	285	1445
I-VT-TS						0	260	144	0	404

Table 12: Example improvement potentials

The example table shows, that the three highest improvement potentials (marked in grey) are in the informal integration test (1445), the design review (658) and the code inspection (457). Hence, an improvement of these VAs will pay off most and are the ones that should be considered for the improvement phase.

3.6.2.3 Overlap factor (OF)

The overlap factor (OF) gives answer to the question if an overlap exists between two VAs and if removing the VA would save cost. It is calculated as the difference between what the current costs of an activity VA_x are and what the cost would have been if the activity was removed and the faults were found later instead.

The VA does not necessarily have to be removed completely. It may also be possible to remove the VA only for a certain fault type. This means, that test cases or testing procedures to find this fault type may be removed, if the VA is inefficient in finding this defect type anyway.

Since it is usually not recommendable to remove a VA if it is effective in finding defects, it only makes sense to calculate the overlap factor for VAs that find very few faults. Furthermore, if the IP is high for the VA, it should be tried to improve the VA instead of removing it. In essence, the OF is only calculated for VAs that find few faults and have a low IP.

The overlap factor OF is now calculated as the difference of the current fault distribution and the estimated fault distribution if the VA was removed and the faults were found in later activities:

The overlap factor is calculated as follows:

$$OF(x) = \left(SEC_{current}(x) + (RRC(x) \times FF(x)\right) - \left(SEC_{future}(x) + \sum_{j=x+1}^{n} (RRC(j) \times FF(x) \times p(x,j))\right)$$

Where the function *FF(x)* denotes the number of faults found in VA_x and *p(x,j)* defines the probability that VA_j finds the fault that was previously found by VA_x.

The current costs of the VA are calculated as the sum of the current SEC of the VA and the number of faults found in the VA times the RRC. The future cost are the new SEC (i.e. a reduced SEC if the VA is not completely removed or 0 if the VA is removed) plus the sum over the fault cost as they will be if they are found in the later activities. This is calculated by the total number of faults found in VA_x times the probability that a later VA_j will find it times the RRC of this VA_j.

Since this probability function in reality does not exist, it has to be estimated how many of the faults that were previously found in VA_x are now found in VA_j.

The formula can be simplified as follows:

$$OF(x) = \Delta SEC(x) - \sum_{j=1}^{n} RRC(j) \times \Delta FF(x)$$

Hence, the OF is the difference in the SEC, i.e. the gain from the removed SEC, and the cost incurred by the new fault distribution, i.e. the loss because the faults are found later which in general tends to be more expensive.

If the overlap factor is positive, it means that removing the activity will result in saved costs, since too few faults are found and the VA cost too much.

To exemplify the usage of the OF it is calculated for the defect data from Table 10 for the UT which only finds very few faults (4). Let’s assume these are interface faults and the test cases for testing these types of defects should be removed and would save 20 working hours. As a next step, the new fault distribution has to be estimated, which is illustrated in Table 13. The UT finds now four faults less, since it was removed and cannot find these faults anymore, and they were estimated to be found in CI (1), I-IT (2) and I-VT-TS (1). Considering the RRC data from Table 11, the result of the calculation of the OF is 8h. This means, that by removing the UT, 8h can be saved.

RR	DR	UT	CI	I-IT	I-VT-TS	IT	VT-TS	VT-RB	AR	Operation
-	-	-4	1	2	1	0	0	0	0	0

Table 13: Example difference in fault distribution

Although several calculations are made on larger set of data, the analysis can still be performed easily by implementing the calculations in a spreadsheet or small program.

3.6.3 Improve

Measure
Analyze
Improve
Implement

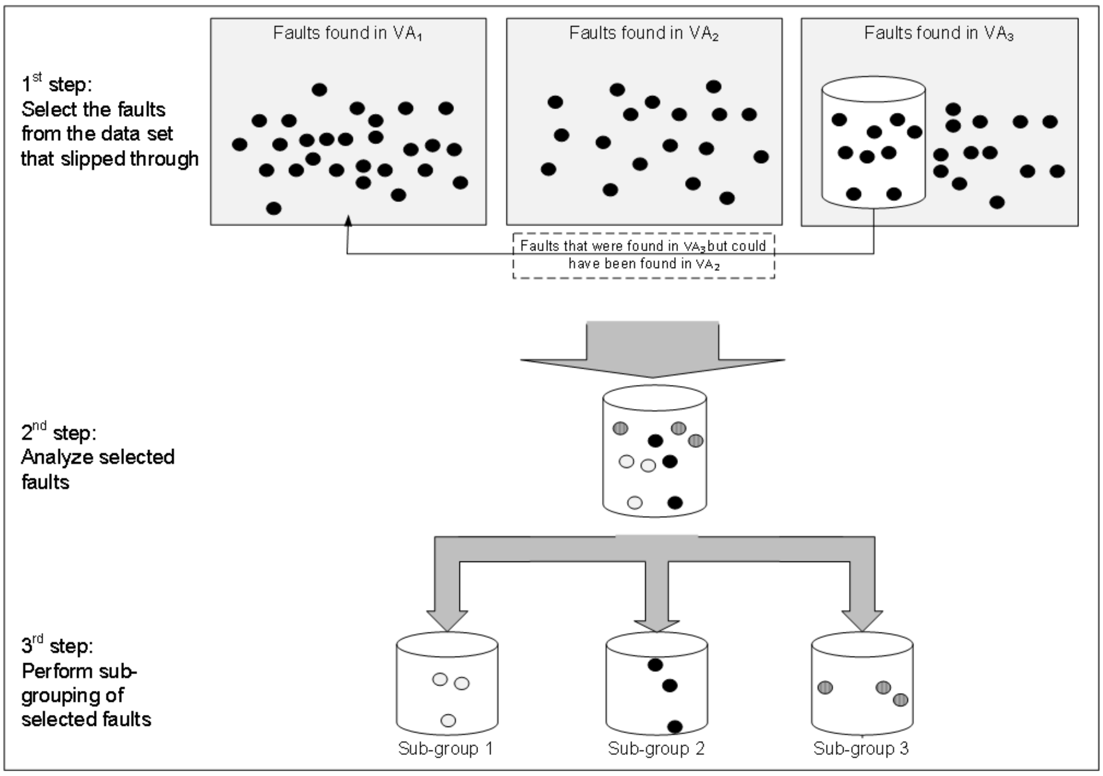
The task of the improvement phase is to develop and evaluate appropriate changes for the VAs that were previously selected in the analysis phase. Hence, change proposals (CP) are developed to improve the VAs with a high IP or a positive OF.

In a first step, the root cause of the problems in the VA is analyzed and changes are developed to overcome these problems. In a second step, the benefits of the CPs are calculated to be able to make quantifiable decisions on which of the CPs should be implemented. Finally, appropriate change proposals are selected.

3.6.3.1 Fault grouping

The VAs that were selected in the previous phase, i.e. the ones with the highest IPs, are further analyzed to identify the problems. This can be the VAs where the IP exceeds a certain threshold or e.g. the VAs with the three highest IPs.

The root cause is identified by analyzing the faults that slipped through the VA. Figure 35 shows an example analysis for a VA_1 where the faults have slipped through to VA_3. As seen in Table 10, this data already exist from the measurement phase. Note that since the IP is considered per fault type, the chosen defects are already of the same type.



1st step: Select the faults from the data set that slipped through
Faults found in VA1
Faults found in VA2
Faults found in VA3
Faults that were found in VA3 but could have been found in VA2
2nd step: Analyze selected faults
3rd step: Perform sub-grouping of selected faults
Sub-group 1
Sub-group 2
Sub-group 3

Figure 35: Sub-grouping of defect data

In a next step, these defects are then further grouped into different subsets according to similarities among the defects. This is done by taking a closer a look at the changes made in the code, design or documentation. The sub-grouping is performed since it is usually difficult to derive improvements solely from the information of the fault type and the VA where the fault could have been found.

3.6.3.2 Root-cause analysis and development of change proposals (CP)

In the last step, change proposals are generated for each of the subgroups. Since the fault type, phase found and phase could have been found are known for the selected defects and the faults were further refined into subgroups according to similarities, enough information about the defects and the problems in the VA is known to draw conclusions and define CPs. The development of the CPs can be done in a brainstorm meeting with experts of the VA. The central question that should be answered in this meeting is how the VA needs to be changed to find these faults that slipped through. Examples of change proposals are:

- Introduction of new (testing) tools
- New check list items in the code inspection sheet
- Modified coverage criteria
- Changed test case selection procedures

Note that although a CP is defined for a specific VA and a specific fault type, the CP can still have an influence on the other VAs or fault types.

Alternatives

The root-cause analysis described in this section, i.e. taking a small group of faults that coincide in phase “found”, phase “could have been found” and fault type and performing a second grouping regarding further similarities, is an easy way to find out the problem and derive change proposals. There exist however more root-cause analysis methods in literature

[AND06]. Examples are the method of *5 whys* and the cause-and-effect chart. In *5 whys*, the root cause of the problem is derived by constantly asking why to the problem and thus getting to a deeper understanding of it. In the cause-and-effect chart, starting from the problem and its effect, the causes are derived and grouped. Thus, the root-cause analysis described in this section may be replaced by another method if needed.

3.6.3.3 Impact analysis

In the impact analysis, the possible benefits of the change proposals are evaluated and quantified. Since not all change proposals have the same benefit and achieve 100% of the improvement potential. Furthermore, it may appear that a change proposal has a large benefit although in reality, it does not. Thus, only the change proposals that actually bring a benefit should be implemented. Another reason is that different change proposals for different VAs or fault types may be contradictive; hence, the quantification can help to choose the right CP.

The benefit of the change proposal is calculated through an improvement gain, which indicates the possible savings that can achieved if the change proposal is implemented. To better understand the origin of the improvement gain, a closer look at the calculation of costs is necessary.

Calculation of costs

As mentioned in 3.2.2.3, the model by Wagner [WAG06] serves as the basis for the cost considerations. Thus, the cost calculations of the Wagner model are described in more detail in the following and the simplifications made to include the cost calculations into the VAMOS framework.

Wagner denotes the function $\theta_A(i,t_A)$ as the probability that the VA_A does not find the defect i within an application time t_A. Hence, 1- $\theta_A(i,t_A)$ is the probability that VA_A finds the defect i within the time t_A.
Thus, the direct cost for a VA, i.e. the cost that can be directly measured, are calculated in the Wagner model as the sum of the setup and execution cost plus the sum over all faults times the removal costs:

$$d_A = u_A + e_A(t) + \sum_i \left(\left(1 - \theta_A(i,t)\right) \times v_A(i) \right)$$

The direct cost of all VAs is calculated according to Wagner as

$$d_x = \sum_{x \in X} \left[u_x + e_x(t_x) + \sum_i \left(\left(1 - \theta_x(i,t_x)\right) \times \prod_{y<x} \theta_y(i,t_y) \prod_{j \in R_i} \theta_y(j,t_j) \right) v_x(i) \right]$$

Whereas the term

$$\left(1 - \theta_x(i,t_x)\right) \times \prod_{y<x} \theta_y(i,t_y) \prod_{j \in R_i} \theta_y(j,t_j)$$

defines the probability that the defect i is found in VA x and was not found in any earlier VA nor was any of its predecessors found earlier. Since we only consider the cost of a defect in the phase where it was actually found, none of the earlier VAs could have found it, thus:

$$\prod_{y<x} \theta_y(i, t_y) = 1$$

Additionally, with the simplification made in section 3.4.2, the predecessors are not considered as separate faults and therefore, if the fault i was found in VA_x, no predecessor could have been found earlier, hence:

$$\prod_{j \in R_i} \theta_y(j, t_j) = 1$$

With the simplification that $u_x + e_x(t_x)$ are grouped together as the setup and execution cost SEC_x and the removal cost $v_x(i)$ are replaced with the regression and removal cost RRC_x, the direct costs over all VAs are calculated as

$$d_x = \sum_{x \in X} \left[SEC_x + \sum_i \big(1 - \theta_x(i, t_x)\big) RRC_x(i) \right]$$

Since in our approach an *a posteriori* analysis is done rather than an *a priori*, the probability function 1- $\theta_A(i,t_A)$ is not necessary. Instead, it is possible to use the relative amount of faults found in the VA_x. Thus we introduce the function FF(x), which denotes the number of faults found in the VA. Therefore, the direct costs are calculated as follows:

$$d_x = \sum_{x \in X} \left[SEC_x + \sum_i FF(x) \times RRC_x(i) \right]$$

The function FF can be further refined to the function $FF_{,j}(x)$ which denotes the number of faults found in VA_x of fault type FT_j. The formula has to be modified then by adding a sum over all defect types j. The direct cost is then calculated as follows:

$$d_x = \sum_{x \in X} \left[SEC_x + \sum_j \sum_i FF_j(x) \times RRC_x(i) \right]$$

Change estimation and improvement gain

The improvement gain (IG) of a change proposal assesses the cost (in working hours) that is saved if the chosen CP is implemented. It is computed by subtracting the current direct cost from the direct cost as they will be in the future.

The current costs are calculated as described in the previous section, by accumulating the SEC of all activities and by summing up the fault cost. Hence, the data from the measure phase are used to calculate the current costs.

$$Overallcost_{current} = \sum_{i=1}^{n} SEC_{current}(i) + \sum_{j=1}^{m} \sum_{i=1}^{n} FF_{current}(i) \times RRC_{current}(i)$$

Since the future costs are not known, it is necessary to estimate the required data. Thus, for each VA it has to be estimated how the SEC, the fault distribution and the RRC will change after implementing the CP. The future cost is calculated equally to the current cost, but by using the estimated data instead of the measured.

$$Overallcost_{future} = \sum_{i=1}^{n} SEC_{future}(i) + \sum_{j=1}^{m}\sum_{i=1}^{n} FF_{future,j}(i) \times RRC_{future}(i)$$

The improvement gain can then be calculated by subtracting the future from the current cost.

$$IG = Overallcost_{current} - Overallcost_{future}$$

It is clear, that if the current costs are higher than the future costs, i.e. the IG is positive, the CP will save working hours and should therefore be implemented.

Since estimating the SEC for each VA and the fault distribution and RRC for each fault type and each VA would require a large effort and greatly increase the uncertainty, several simplifications are made in the following to facilitate the estimation process.

By rearranging the summands and considering the current and the future RRC as equally, the formula can be simplified as shown below.

$$IG = Overallcost_{current} - Overallcost_{future} = \left(\sum_{i=1}^{n} SEC_{current}(i) - SEC_{future}(i)\right) + \sum_{j=1}^{m}\sum_{i=1}^{n} RRC_j(i) \times \left(FF_{current,j}(i) - FF_{future,j}(i)\right)$$

As can be seen, the IG is now calculated by the difference between the current and the estimated SEC plus the difference in the fault cost. Through the combination of the RRC, it is now only necessary to estimate the difference in the fault distribution which is then multiplied by the measured RRC.

The simplification to combine the RRC is made, as it is assumed that the RRC do not change upon introduction of a change proposal. The rationale behind this is that a CP mostly affects the SEC of a VA and the fault distribution, but does neither change the cost to remove a defect in a VA nor the regression cost. Note however, that this simplification holds only true on the supposition that the change proposal has no or very little influence on the RRC of VA_x.

A further simplification can be made, if we assume that a CP only modifies one VA and has only influence on one fault type. This simplification can be made, since a CP is usually developed only for a single VA and any influence on another fault type will only be positive in the sense that at the most more faults are found of another fault type because of the change proposal. Thus, a conservative calculation is made, meaning that at most the estimation of the gain will be too low.

To summarize these simplifications, the difference in the faults found is 0 for every fault type that is not affected by the CP:

$$FF_{current,j}(i) - FF_{future,j}(i) = 0, \forall\ (j \neq t)$$

Furthermore, the difference in the SEC also evaluates to 0 for every VA that is not affected by the CP:

$$SEC_{current}(i) - SEC_{future}(i) = 0, \forall\ (i \neq x)$$

Hence, with the simplifications made the IG is calculated as follows:

$$IG = SEC_{current}(x) - SEC_{future}(x) + \sum_{i=1}^{n} RRC_t(i) \times \left(FF_{current,t}(i) - FF_{future,t}(i)\right)$$

This means, that it is only necessary to estimate the difference in the setup and execution cost for the VA that is changed by the CP and the difference in the fault distribution for the fault type *t* that the CP shall address.

Using the example of the previous section, the usage of the IG is exemplified on a fictitious CP for the activity I-IT, which had the highest IP in the analysis. The CP could be to increase the coverage in the test cases of I-IT to better find interface faults. The cost for these additional test cases are 10h, thus the future SEC is 50.

Table 14 shows the estimated change in the fault distribution. It is assumed that the I-IT will find now 20 faults more that were previously found in later activities. Therefore, the later activities (I-VT-TS, F-IT, F-VT-TS and AR) find now fewer faults.

RR	DR	UT	CI	I-IT	I-VT-TS	IT	VT-TS	VT-RB	AR	Operation
-	-	-	-	20	-10	-3	-6	0	-1	0

Table 14: Example estimation of the changed fault distribution

The calculation of the improvement gain results in 550 hours. This means, that upon introduction of the change proposal, 550 hours can be saved.

3.6.3.4 Define - Selection of change proposal

When the improvement gains have been calculated for all CPs that were developed, it is now necessary to select the CPs which shall be implemented. If the CPs are orthogonal in the sense that they only influence one defect class, several of them may be implemented.

Basically, the CPs with a high improvement gain should be implemented, i.e. the CP with the highest overall savings. Since a change proposal may however require a significant investment (acquisition of tools, training of developers, etc.) it may not always be the best option to go for the CP with the highest IG.

Return on investment

Another way to evaluate the CPs that incorporates the investments that have to be made is to calculate the return on investment (ROI). The ROI computes the amount of money returned from an investment and is defined as follows [RIC04]:

$$ROI = \frac{Benefit - Investment}{Investment} \times 100\%$$

The benefit can be derived from the improvement gain. Since the IG is however calculated in working hours, it needs to be transformed to a monetary value by multiplying the IG with the average cost of one working hour. Further, intangible benefits as e.g. reduced time to market, may be added as well. The investment is every cost that the introduction of the CP will cause, except for higher SEC in the VAs as this is already included in the IG.

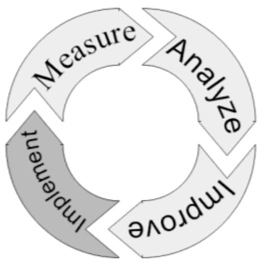
Thus, the ROI is a useful way to decide on which CP should be implemented with respect to the investment that has to be made. Furthermore, it is usually not necessary that the ROI is 100% accurate, but enough to know if it is positive, break-even or negative to make a useful decision [SOL04].

Lowest investment

Another alternative is to go for the lowest investments. This may be interesting for companies that cannot spend a lot of resources on improving the processes.

Apart from the selection of the CP with the highest IG, it should be considered if phases that have a positive overlap factor can be removed. Thus, if the OF for a certain VA is high, it should be considered to remove this VA. As described in section 3.6.2.3, it may also be possible to just remove the VA for a single defect type. Thus the CP would be to adapt the VA (e.g. removing test cases, modifying the code inspection sheet, etc.) so that the VA does not try to find these defect types anymore.

3.6.4 Implement



In the last step of the framework, the selected change proposals need to be implemented. Since the CPs can be very different in their nature, it is not possible to give a general guideline on the implementation of them. The following questions about processes, tools and people can however help to implement the chosen change proposal.

Process

- How does the process need to be changed? Does the workflow or the order of tasks within a VA have to be changed?
- Which process documents need to be adapted?

Tools

- Which tools are required? Do new tools have to be bought or can existing tools be modified and reused?

People

- Which trainings are necessary to teach the developers in the changes made?
- Do new people with special skills have to be hired?

3.7 Alternative options

The framework allows for a parameterization to adjust the required effort that the company has to spend on applying the framework. It may however be necessary to further reduce this effort, if a company has very low resources. Thus two variations of the framework exist which require even lower effort.

3.7.1.1 Alternative 1 – No RRC measurement per fault type

The first alternative removes the RRC measurement per fault type. This means that the RRC is measured for the whole VA and is then divided by the number of faults found in the VA. The advantage of this is that the tracking of the RRC can be greatly simplified, as all cost spent in one VA (except for the initial SEC) can be assigned directly to the RRC. The disadvantage is that the reliability of the analysis is significantly decreased, since the cost for the different defect types are not accurate.

3.7.1.2 Alternative 2 – No fault types

A further effort reduction can be achieved by removing the fault types. This option may be useful if the company does not have a fault classification and does not want to create one. By removing the fault classification, the developers do not have to classify the defects and also do not have to track the RRC per fault type as in the first alternative. This option is therefore the basic FST introduced by Damm [DAM06].

The option has the lowest effort, but also the lowest analysis quality. Additionally, the overlap factor is not useful for this option, since no overlap can exist between different activities because no defect types exist.

4 CASE STUDY AT RUAG

4.1 Analyzed projects

Four projects were considered for this case study and are described in the following. Other projects were not considered since they were either not typical projects or were developed using different software processes and thus are not representative for the study respectively for any process improvement recommendations.

Project A

Project A was a small project with about 50k source lines of code (SLOC) and therefore very typical for the software produced at RUAG. The project was also typical for the current software processes, since it was completed in 2008 and the processes haven't changed much since then.

Project B

Project B was a rather big project with about 180k SLOC. It was developed under a tight schedule and under a modified software process using early prototyping. Therefore it is a rather untypical project for RUAG, but was taken into consideration for this case study since it contains a larger amount of defect data; in particular software problem reports (SPRs).

Project C

The last project that was taken into account for this study was another small project which was still under development under the time of writing this report. Because of this the project was chosen as a representative of the current software development process.

Project D

A forth project was only taken into account for the development of the fault classification. It contains about 300 SPRs which are representative for more complex faults that are usually found in later stages. It was however excluded from the further process analysis since it was already finished in 2002 and the processes that were used back then differ very much from the current processes. Improvement recommendations derived from this data would therefore not have been applicable to the current software processes and thus not be of value.

Note that all projects comprise also a generic boot component which accounts for about 10k LOC. Since this component is usually only slightly adapted for each project, it was not included into the historic analysis, but still used for the development of the fault classification.

4.2 Inputs

4.2.1 Effectiveness data sources

As described in the measurement options model in section 3.5.2.2, the first step was to analyze the existing measurement processes to reason if the effectiveness data is available for each VA. Along with this it was necessary to investigate whether the available data also provides the sufficient quality to be incorporated into the framework.

4.2.1.1 Defect reports

The first data sources to look for are the defect/bug reports, since they usually provide the necessary information or are easy to adapt to collect the required information in the future. Table 15 lists the defect reports that were available for the different VAs at RUAG.

Activity	**Bug reports**
DR	Design Review document
DDR	Code Inspection sheets
UT	---
CI	Code Inspection sheets
I-IT	---
I-VT-TS	---
F-IT	Software Problem Reports
F-VT-TS	Software Problem Reports
VT-RB	Software Problem Reports
AR	Software Problem Reports

Table 15: Bug reports of each VA

All bug reports contain general information about the defect, such as:

- Project name
- Module name
- Version
- Creation date of the report
- Author of the bug report

Apart from these fields, the bug reports differ in the level of detail and the available additional information.

Design review document

The Design Review document is a text document describing the design flaws found by the reviewer. In a table, for each fault that was found the following information is given:

Paragraph	A reference to the corresponding paragraph in the software design document where the fault occurs
Comment / Suggestion	The description of the actual fault in the source code that made the defect occur
Author's response	The action of the author to fix the flaw, e.g. diagram/description updated, ignored or no action.
Status	Status of the problem, e.g. OK or closed

Table 16: Available information of the Design Review document

Code inspection sheets

The Code Inspection sheet contains the interface-definition (.h) and implementation files (.c) of the module that was inspected. The filled-out checklist, the code metrics and the output from the static analysis tool are also attached to the sheet. The actual defect tracking is done in the sheet through annotations at the side of the source code. I.e. the reviewer marks the location in the source code where the fault or flaw was detected and annotates it with a comment describing the problem.

Cause of rejection

The reviewer doing a Code Inspection typically not only writes a comment for faults, but also for suggesting improvements, asking for clarifications or point out minor flaws. Hence, not every comment in a Code Inspection causes a rejection of the code.

Software problem reports

The Software Problem Reports are generated from Rational ClearQuest, a bug and change tracking tool from IBM [CLQ09]. The report contains the following information:

Severity	The severity of the defect (Minor, Major).
Priority	The priority of the defect (Urgent, Normal, Low).
Cause Process	The phase where the defect was introduced (allowed values were taken from the default list of ClearQuest and were not adapted to RUAG).
Specific Cause	The reason for the problem or the software artifact where the defect occurred (allowed values were taken from the default list of ClearQuest and were not adapted to RUAG).
Detected by	The phase in which the defect was found (allowed values were taken from the default list of ClearQuest and were not adapted to RUAG).
Symptoms	A description of the symptoms of the defect. This description is mainly written for the customer.
Cause	A description of the actual fault in the source code that made the defect occur.
Solution	A description of the solution to fix the bug.

Table 17: Available information of the Software Problem Report

RUAG starts to write SPRs very late in the process, specifically after TRR has been reached and thus, for the first time during IT. At this point in the software development process, fixing a defect usually involves several people. Therefore, any action taken out to fix the bug (e.g. redoing the Code Inspection) is listed in a table with the following fields:

- Number of the action
- Status (Open/Closed)
- Description (Description of what was done)
- Result (List of files that were changed)
- Assigned (Person assigned to do the action)
- Closed by (Person who closed the action)
- Date (Date when the action was closed)

4.2.1.2 Further available sources

As can be seen in Table 15, there are no direct bug reports available for the activities UT, I-IT, and I-VT-TS.

Besides the aforementioned obvious data sources, it is also possible to extract the effectiveness information from other data sources that might however be more difficult to analyze.

Commit comments

A further data source is the history from the source code versioning system. Since changes and bug fixes on the code base are typically commented, these comments can be used along with the diff to the previous code version to extract the defect information.

Test reports and test execution logs (STEL)

Due to the ECSS standard RUAG has to document that a test phase has completed successfully. Hence, test reports and test execution logs are written to document that the tests have been accomplished.

Tickets

The web based bug tracking tool CVSTrac [CVS06] was used in certain projects for defect reporting before the start of SPR reporting. Besides a description of the problem extensive information is logged

- Type (Type of the problem, e.g. *todo*)
- Status (Open/Closed)
- Severity
- Priority
- Creator (Person who created the report)
- Created (Date when the report was created)
- Assigned To (Person assigned to fix the problem)
- Closed by (Person who closed the report)
- Last change (Date when the last change was done)
- Due date (Date when the problem has to be solved)
- Version (Version of the artifact)
- Subsystem (Name and version of the subsystem)
- Derived From (How or where the fault was found)
- Fixed By (Person who fixed the problem)
- Cause (Cause of the problem)

4.2.1.3 Quality analysis

As listed in the previous section, several data sources already exist that could serve as measurements for a future VAMOS application and for a first VAMOS iteration on historic data. However, several problems exist with each data source, which lead to the fact that these data sources respectively the current measurements may not be used without modification for a future VAMOS application and make it difficult to perform a historic analysis.

Design review

- The design review mostly consists of faults regarding the documentation of the design rather than the design itself.
- The versions of the design review documents are not always consistent with the design, since it is based on intermediate versions of the design document that are published in the document repository. Thus, an *a posteriori* analysis is very difficult to perform, since it is not always possible to completely understand the defect from the design review without knowing what was changed in the model

Code inspection sheets

- In some Code Inspection sheets the defects which caused the rejection of the code were explicitly indicated in a table at the beginning of the Code Inspection sheet. Because this was done rather irregularly and depending on the person, it was not possible to use this information to exclude defects from the data analysis that were considered by the reviewer as having no influence on the quality of the system.
- Additionally, comments are sometimes made in the code inspections assuming a defect by the reviewer. But a justification is made by the developer that this is not a defect or that the reviewer made a mistake. Thus, not all comments are defects.
- Since modules were sometimes reused, the code inspections were also included from the old projects. This makes it difficult to distinguish whether a code inspection belongs to the current project or not.

Software problem reports

- RUAG starts to collect SPRs very late in the software development process and therefore usually only very few SPRs exist for each project.
- As with the design review, an *a posteriori* analysis is very difficult to perform since the descriptions are typically very short and do not contain the code changes which makes it hard to understand the defect.
- Since the SPRs are mainly used either by the customer to submit a defect report and/or to prove to the customer which defects occurred and that they were fixed, the SPRs are not used for any process improvement. In addition, no formal process or instruction is given on how to fill out the information. This means that in particular the classification about the fault type, the phase where it was found, and the phase where it was introduced differ significantly among the developers and cannot be considered reliable.

Commit comments

- Since the VAs are interwoven (see section 2.5), it is very difficult to map the commit comments that to a specific VA and would require a very high effort. Furthermore, the commit comments only show when the developer submitted the fix of the defect, but not when it was detected.
- It is very difficult to understand and classify the defect *a posteriori* and to figure out where the defect could have been found. It is especially difficult if the analysis/classification is not done by the developer but by another person.
- A very large data set exist (e.g. up to 120 000 commits exist in total for project B, whereas 10 000 commits contain a comment) which needs to be filtered and sampled for an analysis. This filtering however may omit defects and hence, the defect data is not completely accurate.
- Furthermore does the level of detail differ in the comments. Thus, some comments are extensively described, while others consist of only a few words (like e.g. “fix”)

or do not comprise any comment at all. This is due to the fact that no standardized way of writing commit comments is given.

Test reports and test execution logs (STEL)

- As the test reports and execution logs are written at the end of the test campaign, they are only used for documenting that all tests have passed and thus do not contain any information about any defects found in the VA.

Tickets

- The usage of the web based ticketing system was decided by one team to simplify the fault removing and communication among the developers. It was however used only informally and therefore the detail and accuracy of the information differ strongly among the tickets. Furthermore, the standard fields of CVSTrac were used to provide information regarding the defect, but no formal instruction existed on how to fill out these fields. Thus, every developer interpreted and filled out the fields differently.
- It is very difficult to relate a defect to the VA that found it or the VA that could have found it, since the information is missing and the ticket does not contain any changed code.

4.2.2 Cost data sources

The cost data measurement is performed in several spreadsheets for each project and is primarily used for accounting and organizational purposes. The developer basically enters the amount of time spent on each activity.

4.2.2.1 Quality analysis

As with the effectiveness data sources, the current cost data measurement has several drawbacks which are outlined in the following.

Different partitioning

The current cost data is typically split up into *SW Design / SW Architecture, Produce SW Modules*, *SW Integrate* as well as *SW Qualify*. Thus, the cost partitioning is very coarse grained and a further breakdown is needed to retrieve the figures for all atomic activities as listed in Table 5. This however is not possible with the current data since no information is available that would allow such a breakdown.

No separation of SEC & RRC

Since the measurement is used mainly for accounting and organizational purposes, no difference is made between SEC and RRC and the both costs are tracked together in the spreadsheet. This means, that if a bug was found in VT-TS, the time spent on doing unit regression testing is attributed to the unit test cost, the time for the regression code inspection is attributed to the code inspection cost, and so on.

Different layout

The spreadsheets differ among each other in the way of presenting the cost data. E.g sometimes the y-axis contains the names and the x-axis contains the weeks and the developer has to enter the time spent during the week. The differentiation between the activities is done by using several worksheets. In another layout, the names of the developers are on the x-axis and the tasks are on the y-axis, so that the developer enters the amount of time spent on each activity. These different layouts make it hard to analyze the cost data.

Inconsistencies

Since no formal cost measurement process exist, the cost data contain inconsistencies and are not completely accurate. E.g. the work tracking is omitted or different developers attribute the same type of work to different assets. Furthermore, in one project, the cost measurement process was only started in the middle of the project so that half of the data is missing.

Thus, the current cost measurement process is not applicable to VAMOS in its current form and needs to be adapted. Additionally, a historic analysis is not possible with the current cost tracking spreadsheets due to the inconsistencies and inaccuracies.

4.3 Quality Gates

4.3.1 Fault classification

Before conducting this study, the company did not use an internal defect classification. Defects were either not classified or classified using the default classification scheme from the vendor of the bug-tracker in use. This was of course not adapted to the company's domain or the framework. Hence, a defect classification had to be developed that was tailored for the framework and the domain of embedded aerospace applications.

Therefore, the fault classification was created from scratch using the process as described in section 3.5.1.2.

4.3.1.1 Initial fault classification

From the comparison of the existing defect classifications in Table 8, an initial classification was derived that contained classes for the assumed defects and is shown in Table 18.

Fault Type	**Description**
Assignment	Assignment and initialization errors.
Understandability	Documentation, code comments, unnecessary code.
Threading/ Concurrency	Defects in the management of shared resources.
Function/Interface	Requires a formal design change.
Hardware Interface	Problems related to the interface between hardware and software.
Timing/ Performance	Time constraints are not fulfilled; performance is not sufficient
Algorithms	Defects in algorithms, including logical expressions.
Robustness	Boundary checks, error & exception handling.

Table 18: Initial fault classification

4.3.1.2 Iterative refinement

The initial fault classification was then iteratively refined through the following steps:

1. Analysis of code inspection sheets (for the CI activity)
2. Analysis of design reviews (for the DR activity)
3. Analysis of commit comments (for the UT activity)
4. Analysis of software problem reports (for the activities: F-IT, F-VT-TS, VT-RB, AR)
5. Reclassification of code inspection sheets and validation of the fault classification consistency
6. Shadowing and interviews on unit testing (for the UT activity)

Analysis of code inspection sheets (for the CI activity)

It was decided to start the iterations by analyzing the code inspection logs, which proved to be a very good data source. These logs consisted of the source code annotated with the problems found by the inspectors. This should especially help in the first step of the classification development.

To reach a high validity, a random subset of code inspections was chosen (20% of project A and 10% of project B) from two different projects. Then, each defect in every chosen inspection log was analyzed independently by two researchers, which tried to assign a defect type as well as stated how sure they were with their classification and whether they had problems understanding the defect. This was done using a simple ordinal scale with the values *completely sure*, *uncertain* and *don’t know*. In total 622 defects were analyzed, where the classification took between 30 and 45 seconds per defect.

Figure 36 shows for each researcher (A1 and A2) how they classified the defects and how sure they were about the classification. As can be seen, several uncertainties exist for the function/interface, the algorithms as well as the assignment class. Thirteen major issues within the classification were identified in total by performing a root-cause analysis.

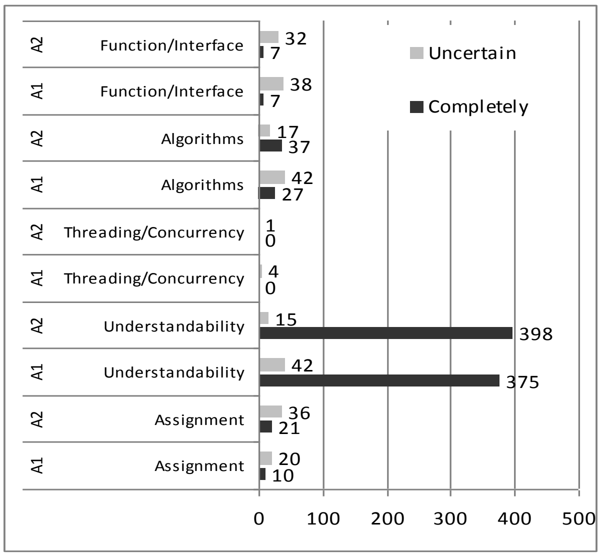


Figure 36: Analyzed faults after the first iteration

One of the results was e.g. that the algorithm and assignment class were collapsed, since it was hard to differentiate the two. Furthermore, the Threading/Concurrency and Timing/Performance were merged and the documentation of the classes was improved.

The revised classification contained then the following classes:

- Algorithm
- Understandability
- Function/Interface
- Hardware Interface
- Timing/Concurrency
- Robustness

Analysis of design reviews (for the DR activity)

As a next step, it was decided to investigate the design review sheets to also include another software artifact than only the source code. Therefore the DR of project B was analyzed which contained a total of 49 defects in three review sheets.

The analysis was performed in the same manner as the code inspection sheets. The main problem here was the distinction between the algorithm and the function/interface class. After discussing the problem, it was decided that the algorithm should only consider the inner parts of the methods while function/interface on the other hand should cover the method signatures, global variables and dependencies across modules.

Analysis of commit comments (for the UT activity)

In the next step, it was tried to analyze the faults found in the UT activity. Since no defect reports existed for this activity, it was decided to evaluate the commit comments from the versioning history. With the problems described in 0, it turned out however that the analysis was rather hard to perform, so that after filtering the comments, 14 defects remained which still couldn’t be related to the unit test activity with 100% certainty. Since these defects were all classified as algorithm defects and no problems or discrepancies between the researchers appeared, no changes were made to the classification.

Analysis of software problem reports (for the activities: F-IT, F-VT-TS, VT-RB, AR)

After the commit comments, the SPRs were analyzed are written after the TRR. The 19 SPRs were taken first only from project A and B. In the comparison, it became apparent that it was hard to decide whether a defect belongs to the hardware interface or not. Since this issue couldn’t be resolved by the researchers, it was decided to clarify this in the quality review with the developers. Another problem that was already observed in the earlier iterations was that several understandability problems existed, which are quite different in their nature and severity for the product. Thus it was decided to split the class intro understandability issues, which can lead to misunderstandings and thus severe defects, and beautification issues, which only consider minor problems like typographic errors and code formatting.

A further problem was discovered regarding the orthogonality of the classes. It was often the case that defects fit into two categories. Since it was considered impossible to make the classes strictly orthogonal, it was decided to introduce a priority ranking instead:

1. Hardware Interface
2. Timing/Concurrency

3. Robustness
4. Interface <-> Algorithm (are orthogonal)
5. Understandability
6. Beautification

Hence, whenever a fault fits into two categories, it is assigned to the class with the higher priority.

Because only very few SPRs existed for the two projects, it was decided to also take into account the SPRs of project D which were about 400 in total. Although the project was already completed several years ago and thus is not representative for the current process, it was still valuable to analyze these SPRs (25% were randomly selected and analyzed), since they contained typical faults, especially regarding the hardware interface and timing/concurrency.

The comparison therefore resulted in a refinement and further documentation of these two classes as well as a modification of the robustness class.

Reclassification of code inspection sheets and validation of the fault classification consistency

As the classification had been revised several times, it was decided to do another analysis of the CI sheets with the updated fault classification and compare the results with the first analysis of the CI sheets. The agreement factor as discussed in section 0 was calculated for both iterations.
In the first iteration, an agreement factor of 0.30 (inadequate) was calculated and a factor of 0.70 (good) was calculated for the iteration with the updated fault classification. Hence, the consistency of the fault classification was now good enough. This proved also the calculation of the agreement factor for the analysis of the SPRs from project D in the previous section, which resulted in a value of 0.71 (good).

Shadowing and interviews on unit testing (for the UT activity)

To gain more evidence about the applicability of the fault classification to the UT activity, it was decided to perform interviews and to do shadowing sessions, which means that the researchers observed two testers while performing unit tests [LET05].

The interviews and shadowing sessions did however not reveal any new problems to the fault classification.

4.3.1.3 Quality review

The fault classification was reviewed in a meeting with four representatives from RUAG. The people were chosen so that a responsible and experienced person for every VA was present. The meeting resulted in several minor changes as well as a change in the priority list, but no changes in the classes were made.

The final classification including the priority ranking is as follows:

1. HW interface
2. Timing/Concurrency

3. Algorithm ←→ Interface
4. Robustness
5. Understandability
6. Beautification

The full fault classification can be found in Appendix A.

4.3.2 Defect measurement options model (MOM-DEMO)

As the quality analysis of the currently available data sources described in section 4.2.1.3 shows, currently no measurement at RUAG does fully satisfy the demanded quality requirements. Hence, no measurement may be used without modification in VAMOS. Therefore, a measurement options catalog was developed to derive the measurements for each VA.

4.3.2.1 Measurement options catalog

The following list describes all measurement options and the manner of how RRC can be measured and sampling achieved.

Spreadsheet

Use a spreadsheet application to track the defect data. The following information need to be collected for each defect:

- Information regarding the module (filename, version) to trace the defect back to the code
- Fault type
- Phase found
- Phase could have been found

Sampling must be done manually. An easy way of sampling can be achieved by using a random number generator tool. That means that the developer requests a random number between 0 and 1 from the tool before entering a defect. If this number is below a certain threshold (e.g. 0.1), the developer has to enter the defect information, otherwise it can be skipped.

The RRC data can be measured directly within the spreadsheet as an extra column or an extra sheet referencing to the defect tracking sheet.

Introduce a lightweight bug tracker

Introduce a new bug tracking tool or web page which should contain fields to enter the following information:

- Information regarding the module (filename, version) to trace the defect back to the code
- Fault type
- Phase found
- Phase could have been found

Sampling must be done manually (see spreadsheet option) if the tool does not support sampling itself. The RRC measurement should be done within the bug-tracker, thus, it should support this option.

Reuse of ticket system

Reuse the existing ticket system described in 0 that was used in earlier projects. The following information will have to be tracked:

- Information regarding the module (filename, version) to trace the defect back to the code
- Fault type
- Phase in which the fault was found
- Phase in which the fault should have been found

Sampling must be done manually (see spreadsheet option) and the RRC measurement can be integrated into the ticket system.

ClearQuest

Use the ClearQuest tool which is already used for the tracking of the software problem reports (SPRs) and tailor it to VAMOS to allow the logging of the necessary information:

- Information regarding the module (filename, version) to trace the defect back to the code
- Fault type
- Phase found
- Phase could have been found

Sampling must be done manually (see spreadsheet option). RRC measurement is already integrated through the actions.

Structured commit comments

Use the commit comments of the source code versioning system for tracking the defects. That means that when submitting a defect, the developer has not only to enter a description, but has also to fill out a short template with the following information:

- Fault type
- Phase found (if the option is used for other phases than unit test)
- Phase could have been found

This requires the developer who fixes the defect to track the defect and make sure that defects that require several changes in different files only are committed once.

Sampling can be done within the versioning tool, by making the template appear only from time to time. RRC measurement might be integrated by adding it to the template. This however only works for unit testing, but not for later phases where more people contribute to the RRC. Otherwise, the RRC measurement can be done within a spreadsheet.

Modify versioning tool

Modify the source code versioning tool to open the web based ticketing system, when the developer commits a bug fix. The following information need to be collected for each defect:

- Fault type
- Phase found (if the option is used for other phases than unit test)
- Phase could have been found

Sampling and RRC measurement can be done similar as with the structured commit comments.

Store test logs

Automatically store the test logs from the test tool. The faults are later on, e.g. after the test campaign is completed or the project is finished, classified with the help of the information from the test logs. The classification needs then to be stored either in the test logs themselves or in another spreadsheet, which may be generated from the test logs. The following information need to be collected for each defect:

- Fault type
- Phase could have been found

Sampling can be achieved by modifying the tool so that it stores only a sample. RRC measurement may be either done within the stored test logs or using a spreadsheet.

Modify test tool

Modify the test tool to store the faults in a database and provide e.g. a web form that allows later classification as soon as the defect was fixed. The form for entering the defect data would need the following information:

- Fault type
- Phase could have been found

Sampling and RRC measurement can be done similar as with the test logs.

Change code inspection sheet

Change the inspection sheet to include a table that lists every defect along with the following information:

- Fault Type
- Phase could have been found

Sampling must be done manually (see spreadsheet option). RRC measurement can be done within the code inspection sheet by having another column in the table to log the RRC cost for each VA.

Write in code inspection comments

The defects are classified within the annotations of the source code in the code inspection sheets. This could happen similar to the structured commit comments by having a template that has to be filled out within the code annotation. The following information need to be collected for each defect:

- Fault type
- Phase could have been found

Sampling must be done manually (see spreadsheet option). RRC measurement can by adding another entry to the template.

Measurement options overview

Table 19 shows a summary of the different properties of the listed measurement options. The table shows how the information is measured with each option. I.e. the information to trace the defect back to the changed code or design, the fault type, the phase where it was found, the phase where it could have been found as well as the removal and regression cost. Additionally it is shown how sampling can be achieved. Here, manually means, that the information has to be entered by the developer whereas automatically means that the information is entered by the tool or is already available through other means. Integrated means that the information can be entered directly while spreadsheet means that it is not possible to track the information within the tool and therefore need to be stored within a separate spreadsheet.

	Source code information	Fault Type	Phase found	Phase could have been found	RRC measurement	Sampling
Spreadsheet	Man. / Integr.	Man. / Integr.	Man. / Integr.	Man. / Integr.	Man. / Integr.	Man.
Lightweight bug tracker	Man. / Integr.	Man. / Integr.	Man. / Integr.	Man. / Integr.	Man. / Integr.	Man.
Ticket system	Man. / Integr.	Man. / Integr.	Man. / Integr.	Man. / Integr.	Man. / Integr.	Man.
ClearQuest	Man. / Integr.	Man. / Integr.	Man. / Integr.	Man. / Integr.	Man. / Integr.	Man.
Structured commit comments	Auto. / Integr.	Man. / Integr.	Auto.* / Integr.	Man. / Integr.	Man. / Integr.**	Auto.
Modify versioning tool	Auto. / Integr.	Man.	Auto.* / Integr.	Man. / Integr.	Man. / Integr.***	Auto.
Store test logs	Auto. / Integr.	Man. / Integr.	Auto. / Integr.	Man. / Integr.	Man. / Integr. or Spreadsh.	Auto.
Modify test tool	Auto. / Integr.	Man. / Integr.	Auto. / Integr.	Man. / Integr.	Man. / Integr. or Spreadsh.	Auto.
Change code inspection sheet	Auto. / Integr.	Man. / Integr.	Auto. / Integr.	Man. / Integr.	Man. / Integr.	Man.
Write in code inspection comments	Auto. / Integr.	Man. / Integr.	Auto. / Integr.	Man. / Integr.	Man. / Spreadsh.	Man.

* when only used for unit testing, otherwise manually
**when used for unit testing, otherwise using a spreadsheet
***if combined with ticketing system, otherwise using a spreadsheet

Table 19: Properties of effectiveness measurement options

4.3.2.2 Rating of measurement options

Table 20 shows the list of measurement options for each VA along with its assumed fixed effort, variable effort, accuracy and the RRC measurement. The rating and selection was made by the two researchers. The final selection was then discussed in a further workshop (see section 4.3.2.3) with responsible persons from RUAG.
Note that the effort ratings do not yet consider the RRC measurement. Thus, an additional effort has to be spent whenever the RRC data cannot be measured within the tool (integrated) but has to be measured in a different tool (in the case of RUAG as a spreadsheet). For simplification reasons this additional effort was added by changing a low effort to medium, respectively a medium effort to high. Since the measurement of the RRC should be done using spreadsheet, it does have only a minimal influence on the fixed effort of the option and therefore only influenced the variable effort.

VA(s)	Measurement Option	Fixed effort	Variable effort	Accu-racy	RRC measurement
AR VT-RB VT-TS IT	Use ClearQuest	Low	Medium	High	Integrated
I-IT* I-VT-TS	Use spreadsheet	Low	Medium	High	Integrated
	Introduce a lightweight bug-tracker	High	Medium	High	Integrated
	Reuse of ticket system	Medium	Medium	High	Integrated
	Use ClearQuest	Low	High	High	Integrated
	Use structured commit comments	Medium	Low	High	Spreadsheet
	Modify versioning tool	High	Low	High	Integrated
	Store test logs	Low	High	Low	Spreadsheet
	Modify test tool	High	Low	High	Integrated
UT	Use spreadsheet	Low	Medium	High	Integrated
	Introduce light-weight bug-tracker	High	Medium	High	Integrated
	Reuse of ticket system	Medium	Medium	High	Integrated
	Use ClearQuest	Low	High	High	Integrated
	Use structured commit comments	Medium	Low	High	Spreadsheet
	Modify versioning tool	High	Low	High	Integrated
	Store test logs	Low	High	Low	Spreadsheet
	Modify test tool	High	Medium	High	Integrated
CI*	Use spread-sheet	Low	Medium	High	Integrated
	Change code inspection sheet	Low	Low	High	Integrated
	Write in code inspection comments	Very Low	Low	High	Spreadsheet
	Use ClearQuest	Low	High	High	Integrated
	Use structured commit comments	Medium	Low	High	Spreadsheet
	Modify versioning tool	High	Medium	Low	Spreadsheet
	Introduce a lightweight bug-tracker	High	Medium	High	Integrated
	Reuse of ticket system	Medium	Medium	High	Integrated
DR	Use spread-sheet	Low	Medium	High	Integrated
	Change design review sheet	Low	Low	High	Integrated
	Use ClearQuest	Low	High	High	Integrated
	Introduce a lightweight bug-tracker	High	Medium	High	Integrated
	Reuse of ticket system	Medium	Medium	High	Integrated

** using sampling*

Table 20: Rated measurement options at RUAG

Table 21 shows the selection that was made for each VA by ordering the options by variable effort, accuracy and finally fixed effort.

VA(s)	Measurement Option	Fixed effort	Variable effort	Accu-racy	RRC measurement
AR VT-RB VT-TS IT	Use ClearQuest	Low	Medium	High	Integrated
I-IT* I-VT-TS	Modify test tool	High	Low	High	Integrated
UT	Modify versioning tool	High	Low	High	Integrated
CI*	Change code inspection sheet	Low	Low	High	Integrated
DR	Change design review sheet	Low	Low	High	Integrated

** using sampling*

Table 21: Selected measurement options

4.3.2.3 Workshop

The measurement options catalogue and the rating of them were performed by the developers. This was done to avoid having several meetings with responsible persons from RUAG and to interfere too much with their daily work.

Since the measurement options may require changes in the processes, it was however not possible for the researchers to decide whether tools and documents may be changed and how much effort can be spend on the measurements.

Therefore it was decided to hold a workshop with the head of software development and two project managers to verify the measurement options, discuss alternatives, get additional feedback and possibly propose other measurement options.

The measurement options were in general perceived positively and some small changes and improvements were proposed. The result of the workshop as well as the final measurement options and adaptation guidelines for RUAG can be found in section 4.3.4.

4.3.3 Setup & execution cost measurement options model (MOM-SECMO)

The problems regarding the SEC measurement mainly affect the historic analysis. Thus, no new way of measurement need to be defined and the spreadsheet can be reused. However, slight modifications have to be made as described in section 4.2.2.

4.3.4 Adaption of the framework to the RUAG software development process

This section outlines how the RUAG software development process needs to be changed to adapt to the VAMOS framework presented in chapter 3. The framework requires that for each VA the cost and effectiveness data is available.

The steps that have to be performed to adapt the framework are listed in the subsequent sections using the following template:

- *Phase(s)* – The phase(s) that need(s) to be changed
- *Adaption description* – A short description of what needs to be changed in the phase in order to incorporate the framework
- *Concrete steps* – The concrete steps that have to be done
- *Motivation* – A motivation why this adaptation is lightweight

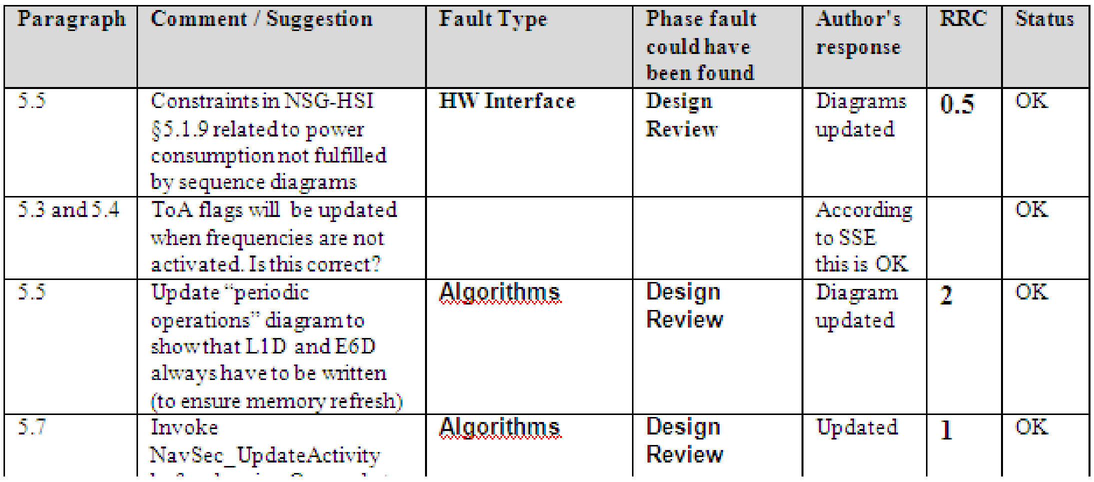
Note, that the motivation only describes why the adaptation to the option is lightweight and why the measurement option facilitates the defect tracking. The process of classifying a defect according to its fault type and the phase where it could have been found can be said to be lightweight in general, since it usually takes not more than 30 seconds to classify a defect (see section 4.3.1). This makes the classification time especially insignificant in the later phases where it can take up to several weeks to fix a defect (including regression tests) and do the defect tracking.

4.3.4.1 Effectiveness data measurement

Design review (DR)

Adaption description

To measure the required information for VAMOS in the design review, the fault information table in the design review sheet needs to be modified. Specifically, columns have to be added for the fault type, the phase could have been found and the RRC data. Figure 37 shows the modified table in the design review sheet.



Paragraph	Comment / Suggestion	Fault Type	Phase fault could have been found	Author's response	RRC	Status
5.5	Constraints in NSG-HSI §5.1.9 related to power consumption not fulfilled by sequence diagrams	HW Interface	Design Review	Diagrams updated	0.5	OK
5.3 and 5.4	ToA flags will be updated when frequencies are not activated. Is this correct?			According to SSE this is OK		OK
5.5	Update "periodic operations" diagram to show that L1D and E6D always have to be written (to ensure memory refresh)	Algorithms	Design Review	Diagram updated	2	OK
5.7	Invoke NavSec_UpdateActivity	Algorithms	Design Review	Updated	1	OK

Figure 37: Change proposal for the design review sheet

Typically, not all problems listed are real defects but rather comments or misunderstandings between the designer and the reviewer and usually lead to a *no action* in the design document. Furthermore, since several design reviews are performed by different reviewers, the same defect may be found and reported by different persons. Because of this, the design responsible has to collect all *true* defects and has to make sure that no bug is classified more than once. For simplification reasons, the design responsible solely tracks the RRC data, even if the requirements document has to be changed. The design responsible therefore has to estimate the RRC data for the change in the requirements.

Concrete steps

1. Append the columns *fault type*, *phase could have been found* and *RRC* to the fault information table of the design review template sheet.
2. Instruct design responsible to collect and classify the defects from the different design reviewers.

Motivation

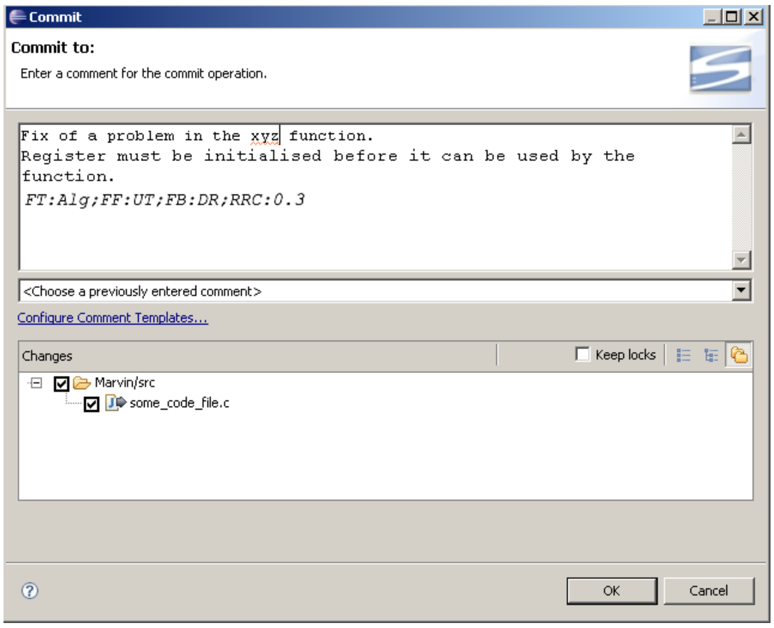
The measurement is lightweight as the defects are already tracked in the current VA and only the template has to be modified slightly to incorporate the three additional fields. Because the design review typically does only find few faults, the classification is not considered to interfere largely with the design responsible's work.

Unit test (UT)

Adaption description

The defect information in the unit test is measured through a hook-up of the version control. This means, that whenever a developer commits a bug-fix, the bug-tracker (i.e. the ticket system) is opened in the web browser and the developer has to fill out a new bug report.
The bug tracker should however only be opened when a bug fix has been made and not when e.g. the developer submits his daily work. This can be achieved by scanning the commit message for the words *bug*, *fix*, *correct*, *error*, *fault*, *defect*, etc.

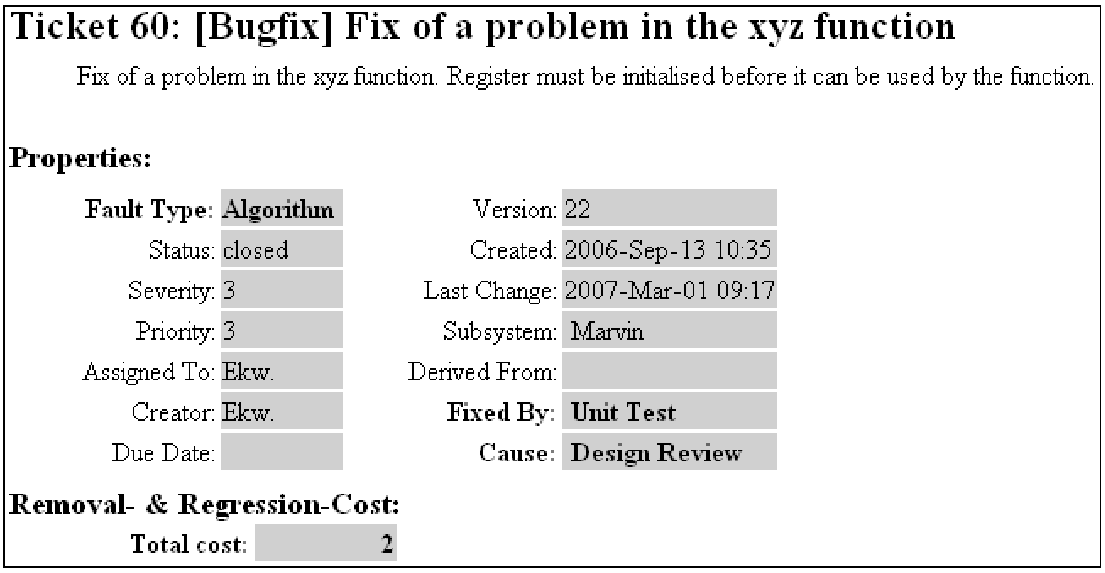
Figure 38 shows exemplary how a commit message could look like using Subclipse [SUB09]. Upon committing this message, the text is scanned for one of the words mentioned above and a new ticket is issued, where the developer enters the fault type, the phase where the defect could have been found, the phase where the defect was found as well as the RRC data.


Commit
Commit to:
Enter a comment for the commit operation.
<Choose a previously entered comment>
Configure Comment Templates...
Changes
Keep locks
Marvin/src
some_code_file.c
OK
Cancel

```
Fix of a problem in the xyz function.
Register must be initialised before it can be used by the
function.
FT:Alg;FF:UT;FB:DR;RRC:0.3
```

Figure 38: Change proposal for the commit mechanism

After providing the information and submitting the report, the bug is stored as shown in Figure 39. Note that further information like the file name(s), the version(s), the creation dates etc. should automatically be filled out by the versioning system upon creation of the bug report.



Ticket 60: [Bugfix] Fix of a problem in the xyz function

Fix of a problem in the xyz function. Register must be initialised before it can be used by the function.

Properties:

Fault Type:	**Algorithm**	Version:	22
Status:	closed	Created:	2006-Sep-13 10:35
Severity:	3	Last Change:	2007-Mar-01 09:17
Priority:	3	Subsystem:	Marvin
Assigned To:	Ekw.	Derived From:	
Creator:	Ekw.	**Fixed By:**	**Unit Test**
Due Date:		**Cause:**	**Design Review**

Removal- & Regression-Cost:

Total cost:	**2**

Figure 39: Change proposal for the ticket system

It should also be possible for the developers to enter all required information already in the commit comment by using a command-line mechanism. Such a command could look like this: *FT:Alg;PF:UT;FC:DR;RRC:0.3*. Which means that the an algorithm fault (FT:Alg) was found in the unit test (PF:UT), could have been found in the design review (FC:DR) and caused removal & regression cost of 0.3h (RRC:0.3). Thus, using the command line mechanism an experienced developer should be able to avoid having to enter the information into the bug tracker.

Similar to the design review the simplification is made that the RRC is only entered by the developer committing the bug fix. This means, that the person committing the defect should estimate the complete cost and enter it, so that no other person has to attach his cost. Furthermore, the developers need to be instructed to commit after each single bug fix, since often several changes are committed at once.

Concrete steps

1. Modify the versioning tool to hook up to the ticket system upon committing a bug fix. This requires the tool to scan the commit message for one of the keywords and for any commands provided. Furthermore, the versioning tool should automatically provide the ticket system with information about the files that were affected.
2. Modify the ticket system to include the fields *fault type*, *phase could have been found*, *phase could have been found* and *RRC*.
3. Instruct the unit testers to commit after each bug with one of the keywords and to classify the defect either in the ticket system or using the command option.

Motivation

A lightweight way of measuring is provided by using the commit mechanism which is already used anyway. By providing the ticket system automatically with most of the necessary information, the developer only has to enter the classification manually. Furthermore, a command line options allows experienced users to classify defects in a fast way.

Code inspection (CI) and detailed design review (DDR)

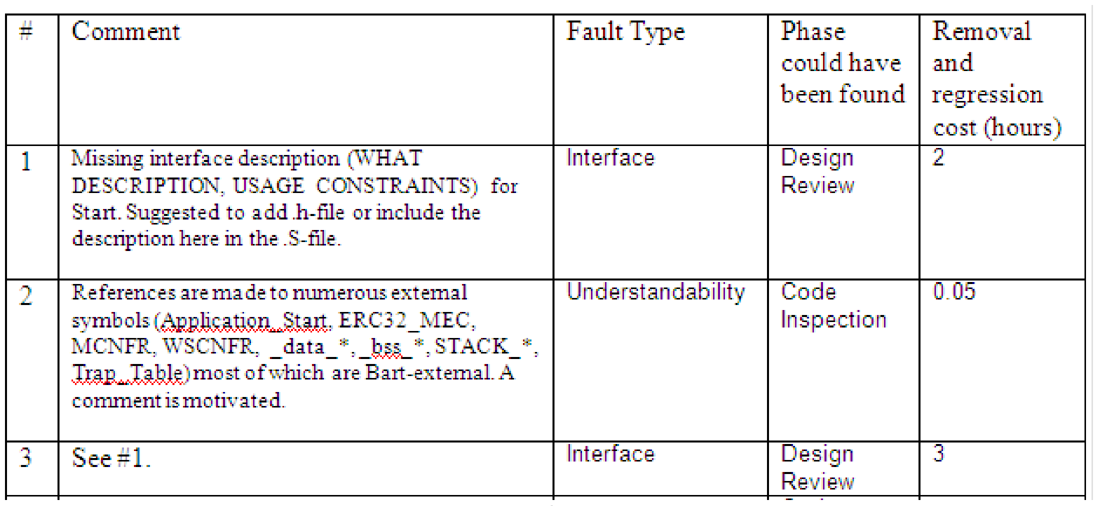
Adaption description

Similar to the design review sheets, the effectiveness and RRC data is measured within the code inspection sheet. Thus, a table with the defect information has to be added to the code inspection template sheet. The table should contain columns for the following information:

- Fault number – The number of the code annotation to which this fault refers
- Comment – A copy of the comment from the annotation to facilitate the classification process
- Fault type
- Phase could have been found
- Removal & regression cost

Figure 40 illustrates the table with example information. To simplify the fault classification, a macro should be developed that automatically creates and appends the table to the code inspection sheet and already fills out the comment column. Since the code inspection phase finds a large amount of faults/problems, sampling should be integrated directly into this macro. This means, that the macro only selects a random subset of all annotations in the document to be placed into the table.

The classification is done by the person fixing the defect, who also estimates the complete removal & regression cost.



#	Comment	Fault Type	Phase could have been found	Removal and regression cost (hours)
1	Missing interface description (WHAT DESCRIPTION, USAGE CONSTRAINTS) for Start. Suggested to add .h-file or include the description here in the .S-file.	Interface	Design Review	2
2	References are made to numerous external symbols (Application_Start, ERC32_MEC, MCNFR, WSCNFR, _data_*, _bss_*, STACK_*, Trap_Table) most of which are Bart-external. A comment is motivated.	Understandability	Code Inspection	0.05
3	See #1.	Interface	Design Review	3

Figure 40: Change proposal for the code inspection sheet

Concrete steps

1. Append a table with the columns *fault number*, *fault type*, *phase could have been found* and *RRC* to the code inspection template sheet.
2. Develop a macro that upon execution selects a random subset of the annotations and fills the table with them.
3. Instruct programmers to classify the defects in the code inspection sheets and to estimate RRC data.

Motivation

Like in the design review, a lightweight measurement is presented by incorporating it into the existing code inspection process. This way an external defect tracking application is avoided. Furthermore, through the use of macros, sampling is performed to avoid having to classify all defect data and the annotation is copied to the table to facilitate the classification.

Informal test campaigns (I-VT-TS and I-IT)

Adaption description

The data measurement in I-VT-TS and I-IT is done using a hookup of an internally developed set of test scripts called tc95 to the ticket system. Thus, upon a failed test case, the ticket system is opened and a new ticket is created. Information about the test case, the file name and file version should be automatically submitted to the ticket system in order to facilitate the bug tracking.

The defect is tracked by the person of the validation team executing the test cases. The classification of the defect is then done by the person responsible to fix the defect. As with the previous options, the RRC is estimated.

Concrete steps

1. Modify the test tool to hookup to the ticket system when a test case fails. This hookup should also include automatic submission of file and test case information.
2. Train the validation team in the use of the ticket system and instruct the development team to classify the defects and log the RRC.

Motivation

As with the other measurement options, the ticket system is reused and thus no extra tool or measurement process is needed. Furthermore, since in I-IT and I-VT-TS no formal way of communicating the defects is prescribed, the introduction of the measurement has also the advantage that the defects are tracked in a central point.

Formal test campaigns (IT, VT-TS, VT-RB and AR)

Adaption description

Since any fault which is found after TRR (i.e. in IT, VT-TS, VT-RB and AR) has to be documented in a software problem report (SPR), the easiest way to measure the faults is by incorporating the necessary information into these SPRs. The forms for writing the defect report in ClearQuest already comprise a *Cause-process* and a *Detected by* field that can be reused to classify the phase the defect could have been found and the phase where the defect was actually found.

Concrete steps

1. The classification in ClearQuest is still the default classification, therefore the classification has to be changed to the one outlined in section 4.3.1.
2. The developers have to be trained in the classification of the defects.

Motivation

The change in the ClearQuest classification requires only a change of the names in the ClearQuest administration panel, which can be done easily by the administrator.

The process definition can be done by briefing the developers in a regular meeting about the defect classification and the use in ClearQuest.

4.3.4.2 Cost data measurement

Adaption description

Use the existing spreadsheet measurement to measure the SEC data. The spreadsheet needs to be modified to include all the VAs outlined in section 2.5.3. Furthermore it is necessary to instruct the team to measure the SEC data separately from the RRC.

Concrete steps

1. Modify the spreadsheet to include all the VAs from section 2.5.3.
2. Train the developers to separate the SEC from the RRC measurement.

Motivation

Since RUAG already performs a spreadsheet based work-tracking where every developer enters the time spent, this can be reused for the SEC measurement. Here, the spreadsheet needs to be only modified slightly, by changing the list of VAs, so that it can be used in VAMOS.

4.4 Framework Iteration on Historic Data

A first iteration of the framework was performed on historic data, since an application of the framework on a current project was not possible due to the fact that the framework was still under development and all projects at RUAG were already in an advanced state at the time of this research.

As the reference project for the historic analysis, it was decided to take project B (see section 4.1). The reasons for choosing project B were that it was the biggest project and contained therefore the most data. In particular, it contained 14 SPR of the later phases like IT and VT-TS. Although this is still a low number, project A contained even less SPRs and project C was not yet completed and therefore didn’t contain any at all. Thus, choosing one of the other projects would result in no defect data at all for these late phases.

Performing the historic analysis on the combined data of all three projects was not considered useful, since the data quality and quantity among the projects differed too much.

The analysis, i.e. the classification of the defects in the different VAs was performed by the two researchers. Both researchers classified the selected defects independently from each other according to the fault classification developed in section 4.3.1. In a second step, problems and differences in the classification were discussed and resolved to have a reliable set of classified defect data.

4.4.1 Measure

Since the analysis was done on a historic project, the data had to be extracted from existing data sources. In the following sections, the defect data and cost data are described which are then used for the analysis.

4.4.1.1 Effectiveness data

Design review (DR)

The design review was performed by three different reviewers. The historic analysis was done by evaluating the defects reported in these three review documents.

Figure 41 shows the faults found in the design review. As can be seen, the most faults were understandability and beautification. Although several interface faults were found, it was expected that the interface faults would by far build the largest defect group in the DR.

Furthermore the DR found several algorithm faults, whereof some of them slipped through from the requirements review.

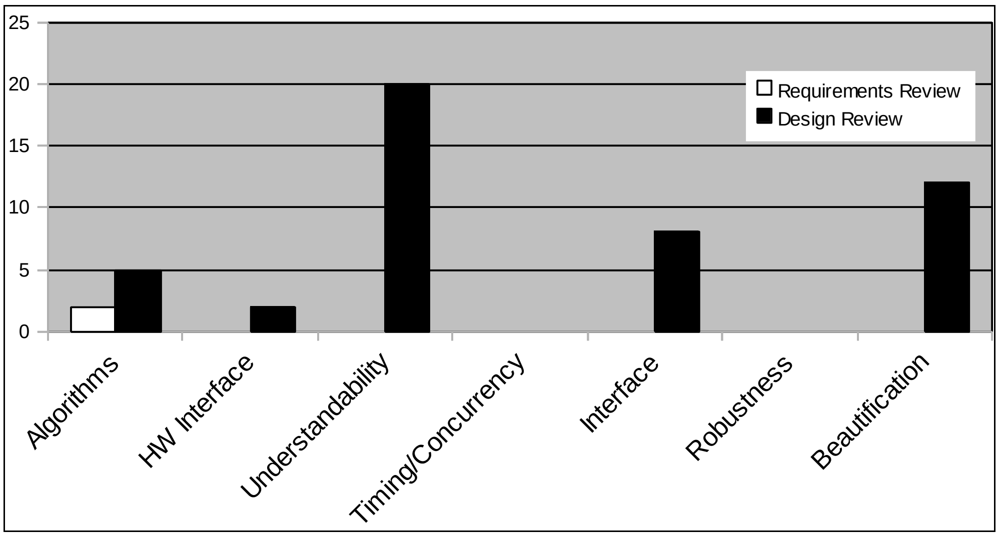


Figure 41: Historic analysis – Faults found in design review

Unit test (UT)

No defect data existed for the unit test activity except for the commit comments (see section 0) which proved however to be difficult to analyze (cf. section 0). Therefore it was decided to perform interviews and conduct shadowing sessions [LET05] in which the researchers observed two developers while they performed the unit testing.

The analysis revealed that in the unit test phase typically one defect per module is found, which is usually an algorithm fault or a robustness fault. This means that for a typical sized project, about 30 faults are found in the unit test activity.

Since the shadowing sessions didn't give a lot of results and the interviews only gave information about typical faults, it was not possible to judge about in which phase the defects could have been found.

The full summary of the interview and shadowing sessions can be found in Appendix B.

Code inspection (CI) and detailed design review (DDR)

The detailed design and the code inspection were considered together. This was necessary due to the fact that the detailed design was done basically as a code inspection on the interface (header) files and it was therefore not possible to distinguish the detailed design review and the code inspection.

One problem that appeared during the classification of the defect data was the distinction of whether a fault could have been found in DR or in DDR (which in essence means the CI activity, since these two phases were considered together). The following table shows therefore the distinction that was made to differentiate these two phases. The left column shows the types of faults which could have been found in the design review while the right column shows the faults that could have been found in the detailed design review respectively code inspection.

Design review	Detailed design review
Method names	Method parameters
WHAT* description (for class & methods)	Return values
Dependencies	Usage constraints
	Implementation constraints
	HOW** description
	Types
	General descriptions

** Description indicating the purpose of the class or method, i.e. what it is supposed to do*
*** Description indicating how the purpose is achieved, i.e. by exemplifying the algorithm*

Table 22: Distinction between DR and DDR for the phase could have been found attribute

Figure 42 shows the faults found in the code inspection sheets. As can be seen directly, the most defects that have been found in the CI are understandability defects. Additionally a lot of beautification defects are found. Almost none of these faults were a slip-through, i.e. almost all understandability and beautification faults could only have been found in CI. This usually means that no problem exists. However, because of the large number of defects it might be desirable to reduce this number of defects by developing means to prevent these defects from occurring.

More important and interesting than the understandability and beautification defects is however the large number of FST in the interface faults from the DR to the CI. This gives already a first indication that the design review needs to be improved.

Another significant fault slip through exists from the UT for algorithm and robustness defects.

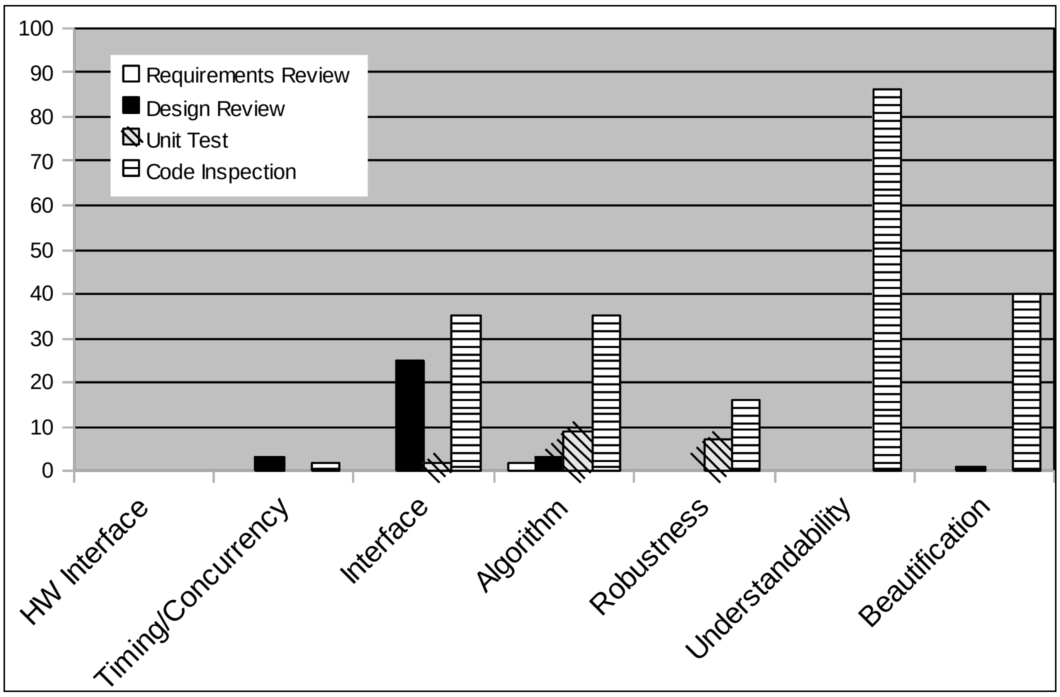


Figure 42: Historic analysis – Faults found in code inspection

Informal test campaigns (I-IT and I-VT-TS)

As described in section 4.2.1, no data sources existed for measuring defects in the VAs IT and VT-TS. Although it is theoretically possible to extract the fault data from the commit comments, it was not considered practically feasible, as the defects in these phases typically span multiple code files and are more complex. This makes it almost impossible to understand the defect *a posteriori*. Hence, the I-IT and I-VT-TS were not considered for the historic analysis.

Formal test campaigns (IT, VT-TS, VT-RB, AR and X-CI)

For the later phases after TRR, only the aforementioned 14 SPRs existed which contained defects for the phases I-IT, VT-TS, VT-RB, AR and an externally conducted code inspection (X-CI). It was necessary to combine the VAs VT-TS and VT-RB in this analysis because the SPRs that occurred in one of these phases were both marked as qualification test, and thus couldn't be differentiated.

Figure 43 shows the defect analysis for these phases. The chart shows the faults that were found in IT, VT-TS as well as in X-CI (on the x-axis). In contrast to the previous charts, the fault types were omitted this time because only very few defects (respectively defect reports) exist for these VAs. The phase where the defect could have been found is indicated by the different patterns (on the y-axis). E.g. one fault was found in IT that could have been found already in DR and four faults in IT could have already been found in I-VT-TS.

The activities I-IT and I-VT-TS were considered the same for the classification of the *phase could have been found* attribute. This was necessary because it could not be differentiated by the researchers if a defect could have been found in I-IT or I-VT-TS.

The most important observation that can be made is that quite a large number of faults slipped through from the design review. Another significant FST occurs from the informal activities to the formal activities, i.e. from I-IT respectively I-VT-TS to the formal IT and VT-TS.

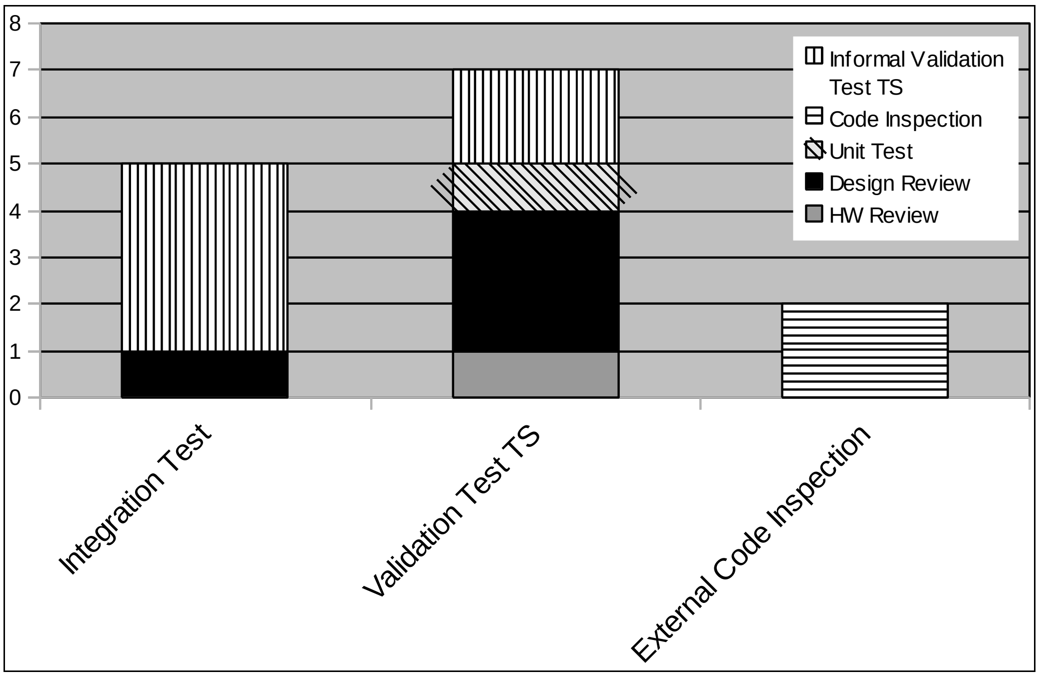


Figure 43: Historic analysis – Faults found in SPRs (i.e. IT, VT-TS and X-CI)

4.4.1.2 Cost data

Since no adequate data source was available for the cost data, it was decided to get data through expert estimation. The cost data was estimated by two domain experts, namely the head of software development and a software project manager, for a typical sized project of 15 000 man hours and about 30 modules.

Table 23 shows the cost data for the VAs. For each VA, the SEC and the RRC was estimated in man hours. Since the framework requires the RRC to be measured for each fault type and it was considered impossible by the domain experts to accurately estimate the cost for each type, a simplification was made. The experts should rate the relative cost for each defect type in each VA on an ordinal scale consisting of the elements *Low*, *Medium* and *High* respectively *Not Applicable* if it was impossible to judge. E.g. the average RRC cost for DR are four hours, but the cost to fix a *Beautification* fault was considered as *Low*, stating that it is usually fixed in less than four hours.

			Beautification	HW-Interface	Timing / Concurrency	Robustness	Interface	Algorithm	Understandability
VA	SEC	RRC	*Low (L) / Medium (M) / High (H) / Not Applicable (NA)*						
DR	100	4	L	M	H	M	M	NA	L
DDR	150	4	L	M	H	M	M	L	L

CI	450	4	L	M	H	M	M	L	L
UT	1200	4	NA	M	H	M	M	L	L
I-VT-TS	3000	30	NA	H	H	H	M	L	M
I-IT	200	30	NA	H	H	H	-	L	M
IT	50	30	-	-	-	-	-	-	-
VT-TS	100	80	NA	H	H	H	M	M	M
VT-RB	100	80	NA	H	H	H	M	M	M
AR	40	80	NA	H	H	H	M	M	M
X-DR	100	10	L	M	H	M	M	L	L
X-DDR	100	10	L	M	H	M	M	L	L
X-CI	100	10	L	M	H	M	M	L	L
X-VT	100	80	NA	H	H	H	M	M	M

Table 23: Estimated SEC and RRC per VA and FT

The historic analysis is then performed by calculating the RRC for each fault type using the ordinal scale. By assigning a numeric value to each scale value, e.g. 0.5 to *Low*, 1 to *Medium* and 2 to *High*, the RRC is calculated for each fault type by multiplying the actual RRC with the coefficient.
Table 24 shows an example of the calculation done for three fault types for the DR activity.

Fault type in DR	Normal RRC	Coefficient	Calculated RRC
Beautification	4h	0.5 (Low)	2h (4h * 0.5)
HW-Interface	4h	1 (Medium)	4h (4h * 1)
Timing / Concurrency	4h	2 (High)	8h (4h * 2)

Table 24: Calculated RRC values

Note that this simplification can influence the analysis quite significantly and results gained by the analysis should therefore be used with care for any improvement recommendations and rather be seen as indicators (see also 4.4.5).

4.4.2 Analyze

Using the effectiveness and cost data from the previous section, the IPs are calculated. Because of the few faults found in the fault categories HW interface and Robustness, the IP was not calculated for these fault types. Furthermore, no IP was calculated for understandability and beautification faults, as there was no or a very low FST for these groups.

Thus, only the IPs for the algorithm, interface and timing/concurrency fault types were calculated as shown in Figure 44. The DR stands out with a very high IP for interface faults. Furthermore the DR has a significant IP for the algorithm and timing/concurrency defects. This strengthens the earlier indications that the DR is one of the phases on which to focus the improvements. Another relevant IP exists for the algorithm faults in I-VT.

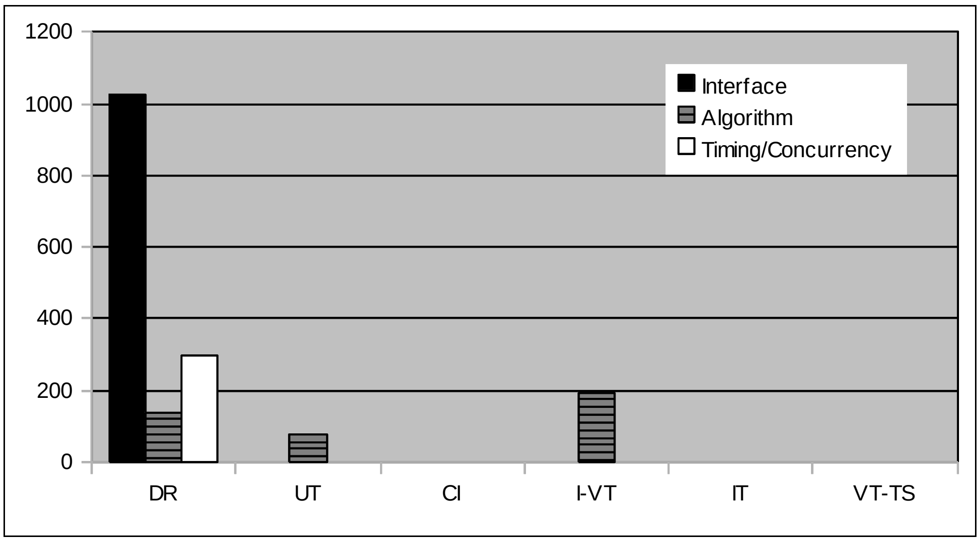


Figure 44: Historic analysis – Calculated improvement potentials

4.4.3 Improve

As the analysis phase revealed, the DR has the highest improvement potential for interface faults. Thus, the interface faults from the code inspection sheets and the SPRs which could have been found in the DR are further analyzed according to the root-cause analysis process described in section 3.6.3.2. Table 25 shows the sub groups that were built by the two researchers and the number of faults that belong to each subgroup as well as the possible savings that can be achieved if the faults are found in DR.

Group description	**# of faults**	**Possible savings (in hours)**
Missing interface description	6	12
Incorrect interface description	10	20
Unnecessary operation/constant	8	16
Missing constant/structure Item	2	80
Compiler settings	1	78

Table 25: Sub-grouping of the interface faults in DR

From the table, the grey marked sub-groups were selected to derive change proposals. The missing interface description can be prevented from reappearing by having automatic checks for empty WHAT descriptions in the design. The unnecessary operations or constants can similarly be checked by an automated tool.

In the next step, the improvement gains are calculated for these two change proposals. Therefore, the 14 errors will not appear in future projects, accounting for a total saving of 28 hours. Note, that since the only 10% of the code inspection sheets have been analyzed, the real number of defects has to be extrapolated and hence the savings as well. Therefore, a total time of 280 hours may be saved by introducing the change proposals.

Furthermore, as indicated in the measure section, the high number of understandability and beautification faults leave room for optimizations. This is especially true, since these faults are usually non-critical issues which may be prevented easily from reoccurring. E.g. the majority of the beautification faults may be prevented by using a spellchecker and a code beautifier. In addition does a large amount of the understand ability faults result from not complying to the coding standard, which may be prevented through the use of tools that automatically check for coding standard compliance.

Although these faults are typically easy to fix and therefore it is not very likely that they will result in a lot of savings, these change proposals will still have the benefit that the inspectors are left from the burden to check for these *simple* faults and can therefore concentrate more on the actual and more severe faults.

4.4.4 Implement

This phase was skipped for the case study, since it was out of the scope of this research.

4.4.5 Threats to validity

Several threats to validity exist for this case study. The biggest threat is that no adequate data sources were available for the use of the case study. As outlined in section 4.2.1.3, several problems exist with the data sources and the historic data in terms of accuracy or quality. Additionally, no effectiveness data sources existed for the informal activities & the unit test and the cost data had to be estimated completely.

Since the fault classification had been developed in this study, another problem was that all defect data had to be classified according to the defect type and phase “could have been found” by the researchers. This always carries the risk that the defects are misunderstood by the researchers and therefore classified wrongly. Furthermore, assumptions had to be made when judging in which VA the fault could have been found.

The results gained from the study, i.e. the problem analysis and the change proposals should therefore be only used as a first indicator for problems in the process.

The case study shows however the principal applicability of the framework to an industrial setting. The problems in the design review were e.g. confirmed by several members from the software development team during a presentation of the VAMOS framework.

5 AGILE APPLICATION AT SSC

This section discusses the applicability of the framework at the Swedish Space Corporation (SSC) in Solna, Sweden. The company was chosen as a representative for companies that have a different development approach than the traditional V-model approach used at RUAG. SSC uses agile principles such as SCRUM [SCH95] and test-driven development (TDD) for developing software. Furthermore, SSC makes heavy use of model-driven development.

The evaluation was done in a three-day's workshop at SSC, where the software development process of SSC was analyzed and the applicability of the VAMOS framework was discussed with SSC employees. The framework was first presented to two developers and one project manager in an initial meeting to get early feedback. The development processes and available data sources (i.e. existing defect measurements) were then further analyzed and the applicability of the framework evaluated. In particular it was discussed which problems SSC faces compared to RUAG and how the framework could help to overcome them and how it could be implemented at the company. In a final meeting the findings were presented and discussed with two developers (one different from the first meeting) and the project manager.

Note that due to the short time frame, it was not possible to perform a full analysis of the software development processes at SSC and how the framework can be applied respectively how it needs to be modified for the use in the company. The discussion presented in the following sections should therefore be only taken as a first indicator on how VAMOS can be applied in an agile company and which problems might arise. A further analysis and adaption of VAMOS to agile environments is still subject to future work.

5.1 Scrum

SSC uses the Scrum software development process, which is an enhancement of the iterative/incremental development approach [SCH95]. In Scrum the software is iteratively developed in so-called *sprints* which typically take about 1-4 weeks. In each sprint a certain set of functionality, which is taken from a so-called *backlog* list, is developed and concludes with a release. Another major difference to traditional iterative development is that frequent reviews should be carried out to address risks and changes early on. A review should be held at least at the end of each sprint.

5.2 SSC software development process

SSC develops in sprints of typically four weeks. Within each sprint, a V-model approach is followed. The concrete activities that are performed within this V-model depend on the component that is developed and the stage in which the product resides.

The software is usually divided into basic software (BSW), which consists of drivers and the on-board software (OBSW), and application software (ASW), which comprises components that reside on a higher application level like e.g. the ground control component (GNC).

Although SSC strives for implementing cross-cut functionality from each level in one sprint for the future, the current situation is that in each sprint one or more components from the same level (e.g. the drivers) are developed.

A typical development process for a sprint looks as shown in Figure 45, which depicts the process for the GNC that accounts for the biggest part in a typical software project. Other components like the OBSW are developed similarly.

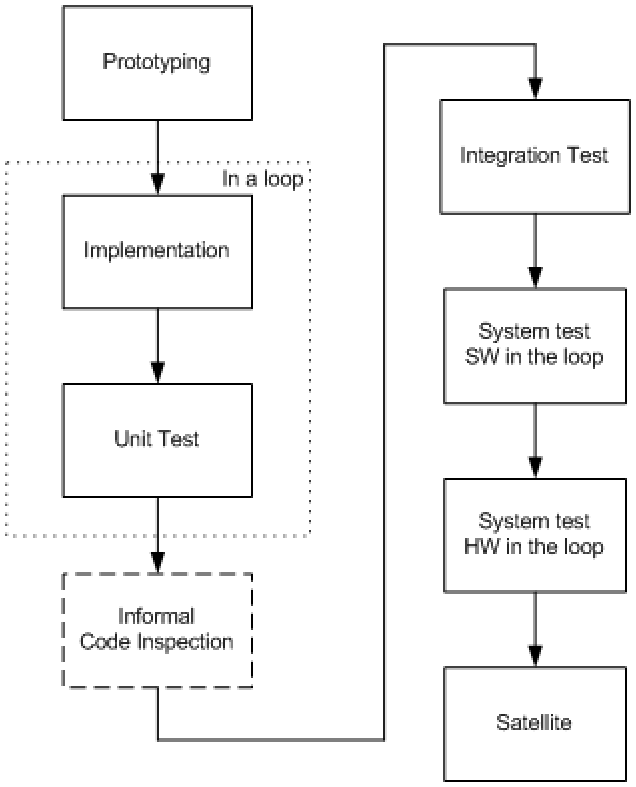


Figure 45: Activities performed within a sprint at SSC

In a first step, the requirements are gathered by prototyping the software and the architecture is outlined. This is not done in a specific order. The OBSW is developed without prototyping. Hence, the requirements are gathered directly and the architecture is developed in parallel to this requirements engineering process.

When the prototype is finished and thus the requirements and the architecture are fixed, the implementation phase begins. The implementation follows a model-driven approach and is developed in Matlab/Simulink. The code is then generated automatically. The models are developed using a test-oriented approach. Hence, the development of the modules and the unit-testing of the model are tightly coupled. The OBSW software on the other hand is developed in C. Since the unit test is done in an open loop, i.e. inserting certain inputs and analyzing the outputs, it is very difficult for SSC to find bugs by only performing unit tests.

The implementation and unit testing phase follows an optional, informal code inspection, which is done depending on the component and module type.

Several integration and system tests follow this activity to run the software in a closed-loop environment. The software is thereby run first in a simulated environment where the outputs of the software are fed into the simulators to produce the new inputs for the software system, thus forming the closed loop. The simulators are then replaced piece by piece with the actual hardware (i.e. original CPU, real sensors, etc.) to test the software in a more realistic environment. At the end the software is tested on the target satellite.

In the integration test, the software is simulated end-to-end. This activity follows the so called software in the loop system test (ST-SIL), which means that the closed-loop is simulated through software (e.g. in Matlab/Simulink).
After that, the hardware in the loop system test begins (ST-HIL). Thereby, the software is run directly on the target processor and as many simulators as possible are removed and replaced with the actual hardware.
In the end, the software is run on the target hardware in the satellite. In this activity, the closed-loop is simulated using the real hardware. The sensors are e.g. fed with currents so that they deliver real, physical data to the system.

In the first few sprints, it is not yet possible to perform directly the whole set of testing activities up until the test on the satellite, partly because the software does not yet have enough functionality and partly because the satellite is not yet available. Thus the software is only tested up until ST-SIL in the first sprints. Only in the later sprints the ST-HIL testing is performed and the testing on the satellite is only done in the last sprints. Hence, in the first sprints the functionality is implemented and tested up until ST-SIL as depicted in Figure 46. Whereas in the later activities the implementation has been completed and only testing is performed in the sprints. The last sprints can therefore be seen as pure verification & validation sprints.

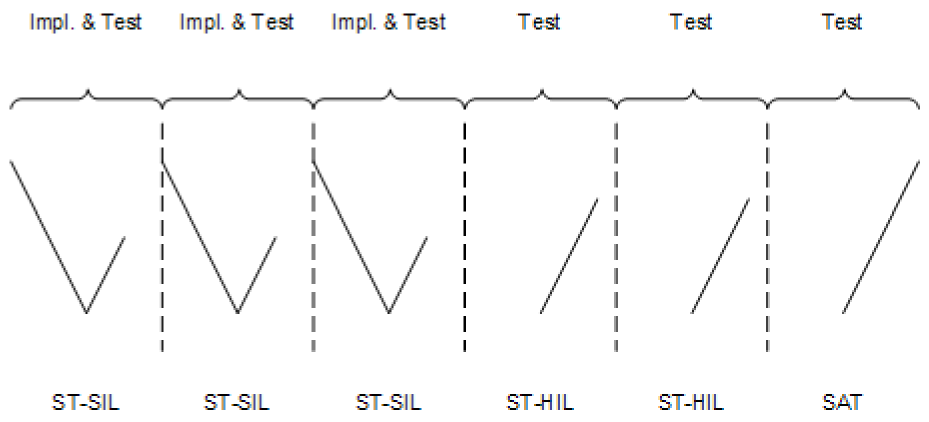


Figure 46: Change of activities during sprints at SSC

5.3 Discussion

5.3.1 Situation at SSC

As discussed in section 1.4 RUAG faces the problem that the later the faults are found in the process, the more expensive they are, as outlined by curve A in Figure 47. In contrast to this, SSC does not explicitly face the problem that the faults which are found later-on are more costly. SSC rather estimates the cost curve to be a linear one with a possible slight slope as illustrated by curve B.

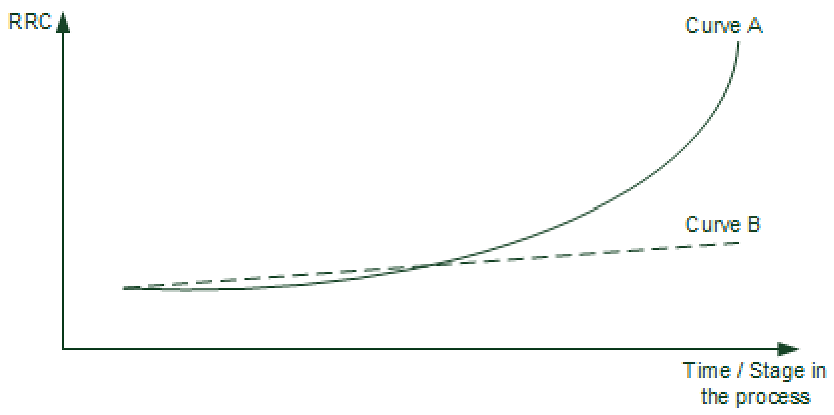


Figure 47: Fault cost in time

Another problem SSC is facing is the loss of knowledge through fluctuation of employees. This is especially problematic in agile environments, since it is tried to avoid extensive documentation.

5.3.2 FST in iterative development

There exist two kinds of fault-slip-through in the scrum development path. The first type is caused within a sprint, since a V-model like process is followed. A fault might e.g. slip through from the unit test to the integration test phase inside a sprint.
The second kind of FST can happen between two sprints, where a fault could have been found in one sprint, but was found several sprints later. In the workshop at SSC, this second FST was considered possible, since the sprints are aligned to scenarios, and it sometimes happens that a defect was not detected within the first scenario, but in a scenario a few sprints later.

5.3.3 VAMOS and MDD

Model-Driven Development does not cause any problems for the usage of VAMOS, since SSC runs the test cases on the model. Thus for the applicability of VAMOS it does not matter whether the bugs are found in the model or in the actual source code.

Regarding the collection of data, one of the biggest drawbacks might be caused by the informal communication that agile development strives for. Thus, it may be difficult to collect the necessary defect and cost data in an agile environment.

5.3.4 Further benefits

Since SSC estimates the costs to find & fix a defect to be more or less constant throughout the process, the company may not gain as much savings through a reduced FST as RUAG. Although this drawback diminishes the reason for introducing the framework within the company, an introduction of VAMOS holds other benefits. These benefits are described in the following and might also be applicable to other agile organizations.

Measurements are introduced

At the time of writing this report, SSC does its process improvement based upon expert opinion. Although this works out well for SSC, the value of improvements could be quantified by the measurements introduced through VAMOS. E.g. SSC currently considers the introduction of model-based testing. Thereby, VAMOS could help to quantify the benefits of the model-based testing.

Furthermore SSC already performs certain measurements, which are however not put into use for process improvement. SSC makes e.g. extensive use of a bug-tracking system and already reports several information. This defect data is however only used to make sure that all defects are fixed and to trace back changes. Thus, without much additional effort, the existing defect and cost data could be used as input for VAMOS to do quantifiable process improvement.

Assumptions are validated

SSC estimates the fault cost to be a rather linear. As FST is a major problem and also literature states that faults found after product delivery are often 100 times more expensive [SHU02], the VAMOS framework could help to validate this assumption or to reveal a hidden FST.

Existing VAs are evaluated

Another benefit of an introduction of the VAMOS framework is that existing VAs are evaluated against each other. E.g. does SSC occasionally perform informal code inspections. This is however not explicitly specified by the software process description, but rather based on the judgment of the developers. An introduction of VAMOS could help to analyze the benefits of a VA to judge if and when a VA should be done or not.

Reduction of the *verification & validation sprints*

As described in section 5.2, SSC uses the last sprints to further test the system. In these typically five to eight sprints no more additional source code is developed. VAMOS might help to reduce the number of *verification & validation sprints* and thus to reduce working hours allowing a faster software production.

Early feedback

Since SSC performs about 20 sprints per project with each sprint lasting maximum 4 weeks, an application of VAMOS could already give early feedback for process improvement after every sprint.

Knowledge is externalized

Since one of the problems SSC is facing is the loss of knowledge through fluctuation, an introduction of VAMOS could help to externalize the knowledge, so that process improvement can go on even if the developers change the department or company.

5.3.5 VAMOS introduction at SSC

An introduction of VAMOS requires SSC to introduce defect and cost measurements to provide the necessary inputs for the framework. The measurement options model described in section 3.5.2 allows SSC to derive and define the required measurement processes. Furthermore, since SSC already extensively uses defect measurements throughout most parts of its process as well as cost measurement, it should be possible to use these measurements for VAMOS.
Additionally, SSC has to develop a fault classification which is adapted to their needs. This can be created following the process described in section 3.5.1.

6 DISCUSSION

6.1 Comparison With Related Work

This section discusses the relation of the VAMOS framework to the related frameworks which were introduced in section 3.2.

6.1.1 Fault-slip-through

VAMOS is mainly based on the concept of fault-slip-through by Damm et al. [DAM06] and as such tries to enhance the concept and to overcome the drawbacks.

VAMOS uses the same measurement data as the FST, i.e. the defect data along with the information where the defect was found and where it could have been found. It describes however in detail how the measurement can be performed and what has to be measured. In particular, the RRC data are further outlined.

The analysis is performed using the formulas of FST without modification. The only difference to FST is that the IP is calculated separately for each fault type. Although FST was used with fault types in [DAM05], they were only used in the analysis to decide on what to change. This was done by improving the fault group that had the most faults. This does however not necessarily mean that these faults were also the most expensive ones. Thus, in VAMOS the IP is calculated for each fault class to overcome this drawback. The disadvantage of this approach is however that not enough fault-slip-through data may exist for each fault type.

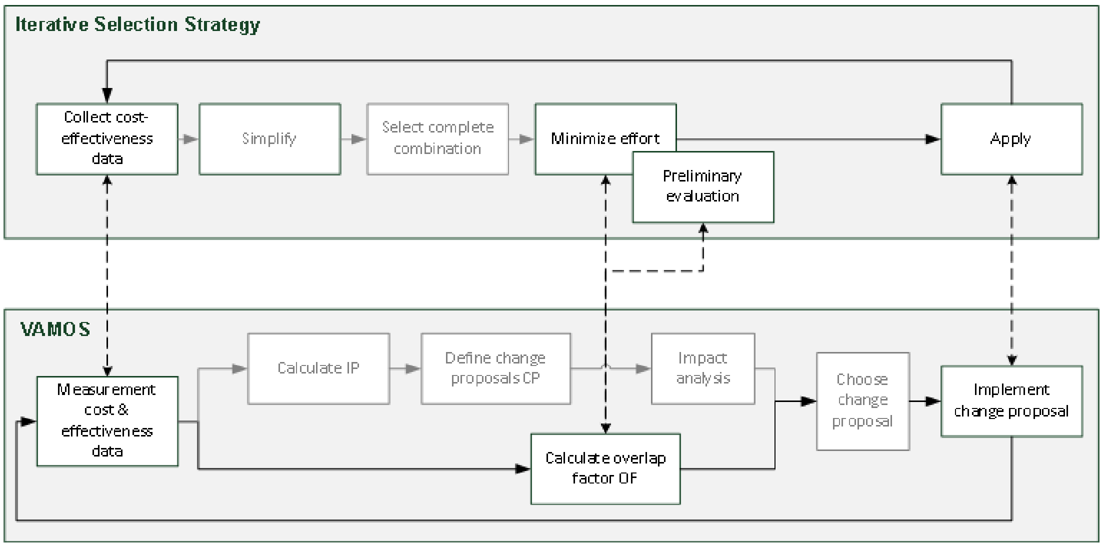
Another drawback of FST is that no means are defined to derive improvements and evaluate them. VAMOS addresses this in the improvement phase.

With the simplifications made in the alternative 2 (see section 3.7.1.2), the VAMOS framework is similar to the original FST as described in [DAM06].

6.1.2 Iterative selection strategy

VAMOS is furthermore based on the iterative selection strategy by Wojcicki and Strooper in [WOJ07] and thus, similarities can be seen directly in the iterative process. Because the focus on ISS is to choose the best VAs out of a set of several VAs, it was designed for a different domain and company context (see section 3.2.2.1). Thus, VAMOS combines the FST with the basic idea of ISS to be more suitable for the space industry. The overlap factor of VAMOS therefore addresses the problem of overlaps between different VAs to keep the benefit of ISS.

In Figure 48 the mapping of the VAMOS framework to the (simplified) activities of the iterative selection strategy is illustrated. VAMOS follows a similar iterative approach as the ISS and also starts with the measurement of the cost and effectiveness data. The activities that cannot be mapped 1:1 between ISS and VAMOS are displayed in gray. The overlap factor can be conceptually mapped to the minimize effort step of the ISS, since in both steps it is tried to remove VAs that find the same types of defects. Furthermore it also already includes an evaluation through the estimation of the benefits and can therefore also be mapped to the preliminary evaluation step. In the last step the changes respectively the new set of VAs (ISS) are applied.



Iterative Selection Strategy
Collect cost-effectiveness data
Simplify
Select complete combination
Minimize effort
Preliminary evaluation
Apply
VAMOS
Measurement cost & effectiveness data
Calculate IP
Define change proposals CP
Impact analysis
Choose change proposal
Implement change proposal
Calculate overlap factor OF

Figure 48: Mapping of VAMOS activities to the ISS

6.1.3 Wagner model

The FST proposed by Damm [DAM06] does not clearly define the cost structure of VAs, that is, which costs related to a VA are taken into account for the FST calculation and which costs are left out. Therefore, a cost structure consisting of SEC and RRC was developed which is based on the Wagner model (cf. section 3.4.2). Furthermore, the calculation of the costs of a VA from the Wagner model was simplified to be used for the framework. The resulting model is used in the impact analysis of change proposals to calculate the overall costs of the VAs (cf. section 3.6.3.3).

6.1.4 Continuous improvement frameworks

Similar to the introduced QIP, Six Sigma, SEI Measurement Process and PDAC (see section 3.2.1) VAMOS follows the typical iterative process. The VAMOS phases define, measure, analyze, improve and implement can conceptually be mapped to similar phases in the said process improvement frameworks.

In contrast to the aforementioned frameworks, VAMOS' focus is not on general applicability, but on a specific context, namely the improvement of the cost-effectiveness of the VAs in the space industry. Hence, VAMOS defines concrete steps, formulas and guidelines to achieve this goal.

6.1.5 GQM

VAMOS relates to GQM in the sense that it can be seen as an instance of a GQM application. The *goal* is to provide a framework for the management and optimization of VAs. This can be refined into the sub goals and *questions* as defined in section 3.1.1. The *metrics* to answer these questions are the IP and OF as defined in section 3.6.2.2 and 3.6.2.3

6.2 Sensitivity Analysis

The model described in chapter 3 contains the cost and the fault estimations as input factors and the improvement potentials, the overlap factor and the improvement gains as outputs. Since the input factors may vary in their accuracy, it is important to find out how the input factors affect the outputs of the model. The primary sources for inaccuracies are caused by effort reduction actions and measurement imprecision.

Effort reduction

As the total number of faults found in a VA can be more than thousand, it may be necessary not to measure all defects to reduce the measurement effort. As shown in section 3.6.1, this can be done through sampling & extrapolating or estimating the data. This obviously reduces the accuracy of the numbers.

Measurement imprecision

Furthermore the data may already be measured incorrectly due to human error or difficulties in the measurement procedures. Especially the measurement of the RRC may lead to problems in later stages of the process, since several people are involved and contribute to these costs. Also, defects are typically fixed in batches, i.e. several defects are fixed at once and are then committed, so that developer will not be able to accurately determine the time they spent on each defect.

Because of these problems, it is important to know more about how the IP, OF and IG is influenced by the cost and effectiveness data. Therefore, a sensitivity analysis (SA) was performed that can determine which factors need more determination and which factors propagate the most variance in the outputs [SAL04].

The sensitivity analysis was performed using the tool SimLab [SIM09] and the model was implemented in Java. SimLab generates random values from a certain distribution, feeds the model with the values and uses the output values from the model to analyze the influence. SimLab provides several distributions. For this analysis however, only normal- and constant-distributions were used. The deviations used in the normal distributions range from 1σ for VAs with a low number of faults to 3σ for VAs with a high number of faults.

The mean values were based on the historic data obtained in the case study. Thus, the sensitivity analysis was performed using the data of project B and contains therefore realistic values. Sampled data was extrapolated and non-existing data, such as for I-VT, was estimated from the experiences gained in the interviews with RUAG representatives. Table 26 shows the FST data and Table 27 the cost data that was used for the SA.

AB/AF	DR	UT	CI	I-VT	IT	VT-TS	AR
DR	50	20	2000	100	1	1	
UT		110	500	500	1	1	
CI			3200	100	1	1	2
I-VT				300	1	1	
IT					1	1	
VT-TS						0	1
AR							0

	50	130	5700	1000	5	5	3

Table 26: FST data for the sensitivity analysis

	DR	**UT**	**CI**	**I-VT**	**IT**	**VT-TS**	**AR**
RRC	4	4	4	30	30	80	80
SEC	300	3600	1350	9000	150	300	120

Table 27: Cost data for the sensitivity analysis

To measure the effect on the IG, it was also necessary to define a fictitious change proposal that changes the fault distribution and the SEC. The change proposal improves the DR with respect to the faults that slip through to the I-VT phase. Therefore it is estimated that 50 faults are found more in DR (i.e. simultaneously are 50 faults found less in I-VT) while the change proposal adds 100 hours to the SEC as shown in Table 28.

	DR	**I-VT**
Faults found	+50	-50
SEC	+100	+0

Table 28: Modified data for the sensitivity analysis

The analysis was performed using the Fourier Amplitude Sensitivity Test (FAST) method on approximately 100 000 samples. The result is illustrated in Figure 49.

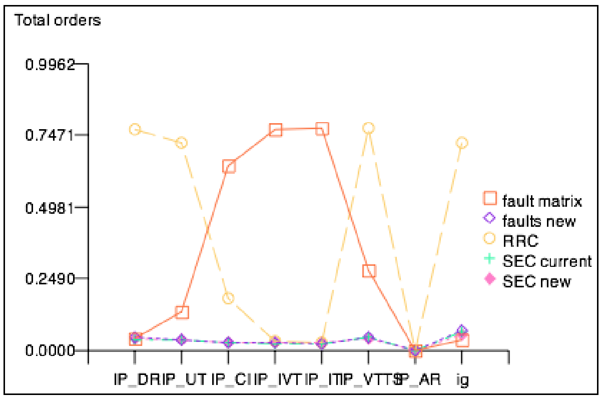


Figure 49: Sensitivity analysis results

Basically, the higher the value of a factor the more influence it has on the output. Thus, as can be seen in the figure, the biggest influences on the IPs have the fault matrix and the RRC. Furthermore has the RRC the biggest influence on the IG.

Analysis results

The analysis of the SA results shows that the RRC has the biggest influence on the IPs whenever many faults slip through. Specifically the RRC of the VA where the faults slip to has therefore to be measured accurately. Whenever few faults slip through, the number of faults becomes more important.

Another surprising result of the analysis is that for the IG, the RRC has a much higher influence on the outcome than the estimation of the future fault distribution.

If the RRC cannot be measured accurately or if doubts exist that the RRC is accurate enough, the IG should therefore be rather used as an indicator of the possible gain than as a hard fact. Another option to overcome this drawback and possibly gain a more accurate result is by calculating the IG with several, different RRC values and use the mean value, best or worst-case scenario instead.

6.3 General Discussion

This section discusses the framework in general. In particular how the thesis objectives are accomplished, what the benefits and drawbacks of the framework are and how the framework was evaluated. Furthermore future work is outlined to point out which drawbacks exist and how they can be overcome to further improve the framework.

6.3.1 Accomplishment of thesis objectives

The goal and sub goals of this thesis project were established in section 1.6.1, along with the research questions and expected outcomes to fulfill these goals. In the following it is described how the research questions and expected outcomes are fulfilled in this thesis project.

6.3.1.1 Research questions

1. How can Fault-Slip-Through (FST) and Iterative Selection Strategy (ISS) be combined into a coherent framework?

Conclusion: VAMOS is an iterative framework like ISS which uses the analysis methods of FST to derive improvements. It combines therefore the strengths of the ISS to reduce the overlap between VAs with the analysis methods of FST to derive cost-effective improvements.

2. Which alternatives exist to the aforementioned models?

Conclusion: Apart from ISS and FST a framework for the comparison of testing activities and formal verification exists [BRA06]. There exists also a more analytical approach for failure prediction [WAG06] (cf. 0). Furthermore do general process improvement frameworks exist like QIP and Six Sigma, which were presented in section 3.2.

3. Which measurements need to be done to support this framework?
4. How can the framework support parameterization to include existing measures and thus, minimize the measuring effort to be real-world applicable?

Conclusion: To support the framework, cost and effectiveness measurements have to be made (cf. section 3.4). The measurement options model (cf. section 3.5) provides a way to include existing measurements of these inputs measures if they fulfill the quality requirements imposed on the measurements. The different tracking options described allow for a minimization of the effort.

5. How can the framework be adapted to a real-world industry setting by only doing minimal changes to the company's processes?

Conclusion: The framework can be adapted to RUAG by applying the guidelines proposed in section 4.3.4. In general, the framework can be applied to a real-world industry setting by following the MOM process (cf. section 3.5.2) to reuse existing measurements or to develop new lightweight measurements.

6. How does the framework perform in a real-world industry setting?

Conclusion: The framework was performed on the data of a historic project (cf. section 4.4) and first improvements could be identified that were confirmed by the team (cf. section 4.4.3). Although an evaluation on a current project could not be made due the short time frame of this research study, the analysis still showed the principle applicability in a real-world setting.

7. How can the framework be adapted to a more agile process?

Conclusion: VAMOS can be applied in a similar way at SSC, by applying it to each sprint (cf. 5.3.2). Furthermore, the measurements can be developed for SSC in the same way as for RUAG (cf. 5.3.5).

6.3.1.2 Expected outcomes

In section 1.6.3 the outcomes were defined that were expected from this research study. The outcomes are fulfilled as follows:

- The description of the framework is given in 3.
- The process guideline for RUAG describing how the framework can be applied to their processes is outlined in section 4.3.4.
- First validation results were obtained in an iteration of the framework on historic data (see section 4.4).
- The list of recommendations for improving the RUAG development process as far as this was possible with the existing data is shown in section 4.4.3.
- The description on the suitability of applying the framework in a more agile environment at SSC is given in chapter 5.

6.3.2 Benefits

This thesis project presented a framework for the management and optimization of the verification & validation. It allows managers to gain insight into the effectiveness (how many defects are found?) and the efficiency (how many defects are found per hour?) of a VA. And based on this, VAMOS allows the comparison of different VAs.

VAMOS provides an iterative improvement process similar to continuous process improvement frameworks like QIP or Six Sigma (cf. 3.2.1) and therefore evidence-based process improvement that are based on measurements. In contrast to the traditional process improvement frameworks like QIP and Six Sigma which aim for general applicability, VAMOS provides concrete measurements and analysis methods to solve the specific problem of optimizing VAs.

Since VAMOS is based on FST, it takes into account cost considerations. Therefore, it does not aim for just improving any VA, but for improving the VAs in the most cost-effective way. Additionally, it helps to find out which VAs do not find the faults they are supposed to find

Furthermore, VAMOS provides an overlap factor to analyze whether two different VAs find the same kind of faults and it is more cost-efficient to remove one of the VAs (either completely or only for a certain fault type). Although, it may not be possible to remove a VA because of the standards, this allows a company to show to its customers what the different VAs cost and what the cost for quality is.

Therefore, VAMOS combines the benefits of the ISS, i.e. the optimization of the selection of VAs, with the benefits of the FST, i.e. the optimization based on the faults that cost the most.

As mentioned before, VAMOS provides concrete recommendations on measurements, analysis and improvements. It allows for deriving measurement processes that require a low effort and in contrast to FST, VAMOS defines ways to derive the improvements from the analysis.

Besides these benefits, VAMOS provides the further benefits as described in chapter 5.3.4:

- Assumptions on the effectiveness of VAs and the FST between VAs can be validated
- Existing VAs are evaluated and compared
- Early feedback and reduction of the verification & validation sprints in agile environments
- Knowledge is externalized

6.3.3 Drawbacks

The biggest drawback of VAMOS is that it requires several measurements. Although the MOM aims for measurements that require a low effort and different tracking options exist, this effort may be further be reduced.

Furthermore does the framework require estimations to calculate the IG and OF. Thus this can lead to inaccuracies.

Although VAMOS incorporates cost-considerations, it does not take into account intangible benefits when calculating the IPs or IGs. E.g. can a VA have a low IP and may therefore not be the focus of improvements, but due to long-term quality improvements, the customer relationship improves and thus the benefit may be higher than initially calculated.

A further drawback is that the framework requires that the developer classifies the defects according to the VA where they could have been found and the fault type. Since this can result in errors, it may be necessary to adjust the fault types and retrain the developers from time to time.

6.3.4 Evaluation

The framework was first evaluated in a workshop with RUAG representatives to gain an initial feedback and improvement recommendations. Furthermore the framework was evaluated on the data of a historic project (see section 4.4). A dynamic evaluation, i.e. an evaluation on a current project, could not be performed due to the short time frame of the research and a lack of a suitable project at the time of this study.

6.3.4.1 Static analysis (workshop)

A workshop was conducted with the head of software development and a software project manager of RUAG. The goal of this workshop was to get an expert opinion on the feasibility and usefulness of the framework. The workshop was conducted partly as an open interview and partly as a questionnaire. In the questionnaire part, the two participants were either given statements where they had to say if they strongly agree, agree, are uncertain, disagree or strongly disagree or had to answer questions.

The representatives agreed with the following statements/questions:

The framework will …

- …give enough information about the problems to derive change proposals
- …help to get a better understanding of the different VAs.
- …help us to compare different VAs.
- …help to reduce the overlap between the VAs
- …reduce effort spent on Verification & Validation

Apart from these statements in which both participants stated that they agree, further questions were answered and are summarized below:

Do you think the measured cost data are accurate?

The participants argued that the cost data might be a problem and that it is not clear which assumptions are ok when estimating and which have a huge influence.

Do you think the measured effectiveness data will be accurate?

The participants answered that it depends on the defect. It could especially be hard in later phases where the developers are not familiar with the early activities. However they estimated that it would be possible and correct in 90% of all defects. The participants also expressed the concerns that VAs are not always in the same order.

Do you think the estimations for SEC, RRC and fault distribution will be accurate?

The participants answered that it is hard to provide more than a guess. Analyzing historic data & pilot projects might help, but not in all cases.

Do you think it will be difficult to use the framework at RUAG?

The participants said that the full framework will be difficult but it will definitely be possible to use some parts of it. They also argued that if it is possible to get the support from the developers and they see the benefit, it is not a huge effort.

How difficult will it be to use the framework at RUAG?

Full framework will be difficult but parts of it definitely. They want to do some kind of measurement.

Conclusion

Although the static analysis has only limited validity because only two persons were involved and it was difficult to judge for them which results a framework application would produce in the future, it still gave useful insights and feedback.

The static analysis revealed that the RUAG representatives are confident that the framework addresses the goals established in the initial meeting (cf. section 1.4). However they also expressed their concerns about the accuracy of data and that they don’t know how inaccurate data will influence the outcomes of the framework application.

6.3.4.2 Iteration on historic data

Because the evaluation on a current project was not possible, it was decided to use the data from a historic project to perform a first iteration (see section 4.4). Although several threats to validity exist to the correctness of the results (cf. section 4.4.5), the iteration served as a proof-of-concept by providing results that were confirmed by the team.

6.3.4.3 Validation of the quality gates

In two workshops the developed fault classification and the proposed measurement options were discussed with RUAG representatives (cf. section 4.3.1.3 and 4.3.2.3). Although the goal of these workshops was not to review and discuss the processes but the classification and the measurement options, the workshop showed that the processes produce useful results since only minor changes were made and the fault classification and measurement options were perceived positively. Further evaluation is however still necessary.

6.3.5 Applicability to other industries

The applicability of the framework to other industries and thus the possibility to generalize the thesis results has not been analyzed explicitly in this research. As the framework was developed for the Space industry, i.e. for industries that have high dependability and reliability requirements and develop embedded software of small size using a traditional process (waterfall or V-Model), it may however be also applicable to similar industries such as the automobile or aviation industry. Hence, a more thorough analysis still needs to be done and is therefore part of future work.

6.3.6 Future work

As the framework has only been validated in a workshop and on historic data, the next step is to perform a dynamic analysis. That is, to apply the framework on several projects over a larger time span to be able to judge whether VAMOS derives useful improvements and whether the defined improvements save time in future projects. Further evaluation needs also to be done in agile companies.

Another future work is to develop tools that support the VAMOS framework application. This can be e.g. web based bug-trackers that implement the FST calculations and give the managers real-time information.

Because the developers have to classify the defects according to the phase where they could have been found and the fault type, it is very likely that this classification is not always performed correctly and thus influences the analysis. Therefore, it may be necessary to constantly check that this classification is performed consistently, e.g. using the agreement factor as described in section 3.5.1.2. This means, that the agreement factor is constantly calculated (e.g. automatically through tools) and if it falls below a certain threshold, the fault classification is revised respectively the people are retrained to correctly classify the phase where the defects could have been found.

In the impact analysis of the change proposal, a simplified version of the Wagner model was used. In particular, all manifestations of one defect in different software artifacts were considered as a single defect. It is therefore necessary to evaluate in how far these simplifications may impact the analysis results and how these simplifications may be different for other companies.

Further work needs also to be done in the improve and implement phases. In the improve phase, the root-cause analysis may be refined to help not only discover where the biggest problems are but also to efficiently find & solve them. In the implement phase, further best practices may be elaborated to support the implementation of change proposals.

6.4 Summary

This thesis presented a framework for the management and optimization of verification & validation activities (VAMOS) which was developed at the RUAG Aerospace Sweden AB in Göteborg.

VAMOS follows an iterative approach similar to Six Sigma or QIP and is furthermore based on the concept of fault-slip-through presented in [DAM06] and the iterative selection strategy presented in [WOJ07]. Thus it combines the benefits of FST, i.e. to take into account cost-considerations to improve only the VAs with respect to the most expensive defects, with the benefits of ISS, i.e. to find the optimal and most cost-efficient combination of VAs.

VAMOS consists of the process steps measure, analyze, improve and implement which are performed iteratively as well as a define process step, in which the goals of the company are set and trade-offs are discussed for tailoring the process steps.

In the measure step, the effectiveness respectively defect data as well as the cost data is measured. The measured data is then analyzed using the concept of FST to find out which faults slip through and which of the faults are the most expensive ones. In the improve phase, adequate changes are then derived to address these problems and are further evaluated with respect to the benefit they produce. In the last step, the changes with the highest potential benefit are then implemented in the company.

Since companies typically have some sorts of cost and defect measurements, but usually not in every phase of the software development lifecycle, it is desirable to reuse existing measurements and develop new ones for the phases that don’t have one yet. VAMOS therefore defines a measurement options model that checks existing measurements if they fulfill the required quality standards (quality gate) and helps to develop new measurements. Furthermore, VAMOS describes another quality gate to help companies develop a defect classification if they don’t have an adequate one yet.

Using data of an historic project at RUAG, the framework was evaluated by performing a first iteration of it on this data and first indications could be given on how to improve the process. Another evaluation was done at the Swedish Space Corporation in Solna, to find out how the framework could be applied in an agile context.

Conclusion

This thesis contributes with a framework to evaluate and compare VAs according to their efficiency and effectiveness and allows for a process improvement that takes into account economic aspects. No such framework currently exists and previous work is either targeted at general process improvement or solves only parts of the problems. Specifically, it expands the concept of FST to an iterative framework that also defines how to derive improvements and calculates the benefit of them. Furthermore it enhances the ISS by not only trying to have a minimal set of VAs, but also by trying to improve existing VAs.

Appendix A FAULT CLASSIFICATION

A.1 Algorithm

A.1.1 Description

All code defects inside a method.

A.1.2 Examples

Name	Wrong code	Corrected code	Description
Condition	If(x<10) { ... }	If(x<=10) { ... }	Wrong

	x = 5;	if (y>x) {
		  x = 5;
		}
			missing

Calculation	int i = a + b;
	x = myArray[i];
		int i = a + b -1;
		x = myArray[i];
			Wrong calculation

"HOW" descriptions of methods	/* * HOW: does A. */	/* * HOW: does B. */	wrong
	/* * HOW: does A. */	/* * HOW: does A then does B. */	missing functionality description
	/* * */	/* * HOW: does B. */	missing HOW comment

Local assigments	function abc() {
	   x = a;
	   ...
	}
		function abc() {
		   x = -a;
		   ...
		}	

	function abc() {
	   x = 5;
	   ...
	}
		function abc() {
		   x = 6;
		   ...
		}	

	function abc() {
	   x = C_ABC;
	   ...
	}
		function abc() {
		   x = C_CDE;
		   ...
		}	

	b(x);	x = 4;
		b(x);
			missing (uninitialzed variable)

Function calls	a();	b();	wrong
	f(a, b);	f(b, c);	wrong parameters
Local defines	#define ABC 0x808	#define ABC 0x909	wrong

	---	#define ABC 0x909	missing

Local typedef/struct	typedef struct {
	  double x;
	  double y;
	} point;
		typedef struct {
		  double x;
		  double y;
		  double z;
		} point;
			wrong

	typedef struct {
	  double x;
	  double y;
	} point;
		typedef struct {
		  int x;
		  int y;
		} point;
			wrong

	---	typedef struct {
		  ...
		}
			missing

	typedef struct {
	  double a;
	  double b;
	  double x;
	  double y;
	} sometype;
		typedef struct {
		  double a;
		  double b;
		} sometypeAB;
		typedef struct {
		  double x;
		  double y;
		} sometypeXY;
			change

Local variables	function abc() {
	  int16 x;
	  ...
	}
		function abc() {
		  int32 x;
		  ...
		}
			wrong type

	function abc() {
	  int16 x;
	  ...
	}
		function abc() {
		  int16 x;
		  int16 y;
		  ...
		}
			missing

Statement order	x = 2;
	b(x);
		b(x);
		x = 2;
			Wrong order with functional influence

	c(x);
	a = 2;
	b(x);
		c(x);
		b(x);
		a = 2;
			Wrong order without functional influence

	c(a);
	b();
		c(a);
		a();
		b();
			missing statement

A.1.3 Distinction to other classes

- Understandability is naming of variables and wrong/missing comments inside methods; also unnecessary code. Everything that doesn’t affect the functionality of the method.
- Interface is concerned with the interface of a module (including the signature of a method) and thus is never concerned with the inner algorithms of a method.

- If the condition is used to check whether a parameter is within the allowed range, the defect is classified as Robustness
- typedefs and defines that are only used locally belong to Algorithm, global typedefs and defines, e.g. from the design document, belong to Interface

A.2 Interface

A.2.1 Description

Defects related to the interface of a module.

A.2.2 Examples

Name	Wrong code	Corrected code	Description
WHAT description (module header and method)	/* * WHAT: does A. */	/* * WHAT: does B. */	wrong
	/* * WHAT: does A. */	/* * WHAT: does A and B. */	missing functionality description
	/* * */	/* * WHAT: does B. */	missing WHAT comment
Includes	#include abc.h	#include def.h	wrong

	#include abc.h	#include abc.h
		#include def.h
			missing

Global defines	#define ABC 0x808	#define ABC 0x909	wrong
	---	#define ABC 0x909	missing

Global typedef/struct	typedef struct {
	  double x;
	  double y;
	} point;
		typedef struct {
		  double x;
		  double y;
		  double z;
		} point;
			wrong

	typedef struct {
	  double x;
	  double y;
	} point;
		typedef struct {
		  int x;
		  int y;
		} point;
			wrong

	---	typedef struct {
		  ...
		}
			missing

	typedef struct {
	  double a;
	  double b;
		typedef struct {
		  double a;
		  double b;
			change

	double x;
	double y;
	} sometype;
		} sometypeAB;
		typedef struct {
		double x;
		double y;
		} sometypeXY;	

global variable	int16 x;	int32 x;	wrong type

	int16 x;	int16 x;
		int16 y;
			missing

Operation signatures	void abc();	int abc();	wrong
	void abc();	void abc(int x);	missing
	int abc(int16 x);	int abc(int32 x);	wrong

A.2.3 Distinction to other classes

- Understandability: rename of typedefs, structs, defines for better understandability.
- Algorithm: inside a method.
- Robustness: const and usage constraints go to robustness.
- Global typedefs and defines, e.g. from the design document, belong to interface. locally used typedefs and defines belong to Algorithm.

A.3 Robustness

A.3.1 Description

Everything that is done in order to prevent failures in abnormal circumstances.

A.3.2 Examples

Name	Wrong code	Corrected code	Description
usage constraints at module/method level	/* * USAGE CONSTRAINTS: trajectory length > 0 */	/* * USAGE CONSTRAINTS: trajectory length >= 0 */	wrong
	/* * USAGE CONSTRAINTS: trajectory length > 0 */	/* * USAGE CONSTRAINTS: trajectory length > 0 trajectory length < 50000 */	missing usage constraint
	/* * USAGE CONSTRAINTS: pointer is not null */	/* * USAGE CONSTRAINTS: none */	unnecessary usage constraint
missing const	int x;	const int x;	

asserts	assert(x>0);	assert(x>=0);	wrong

	/*
	 * USAGE CONSTRAINTS:
	trajectory length > 0
	 */
	function abc(double tl) {
	...
	}
		/*
		 * USAGE CONSTRAINTS:
		trajectory length > 0
		 */
		function abc(double tl) {
		assert(tl>0);
		...
		}
			missing

	/*
	 * USAGE CONSTRAINTS:
	none.
	 */
	function abc(double tl) {
	assert(tl>0);
	...
	}
		/*
		 * USAGE CONSTRAINTS:
		none.
		 */
		function abc(double tl) {
		...
		}
			unnecessary

boundary checks	arr[arra_current] = 2;	if (array_current<=array_max) {
		arr[arra_current] = 2;
		}
			Missing

	If(param1<= 256){	If(param1< 256){	wrong
	If(param1< 256){	If(param1< 128){	wrong
	/* x < 127 */	/* x < 128 */	Comment wrong (wrong range)

A.3.3 Distinction to other classes:

- Interface: Robustness is only about usage constraints.
- Algorithm: Robustness is only asserts and boundary/range checks.

A.4 Timing/concurrency

A.4.1 Description

Issues related to concurrency and compliance to timing requirements.

A.4.2 Examples

Name	Wrong code	Corrected code	Description

concurrency protection	a();	semaphore a(); semaphore	missing
delayed writes	mov %l0, %wim	mov %l0, %wim nop nop nop	missing

- **Timing requirements not fulfilled**
- **Re-entrancy (usage constraints + code errors)**
- **Priority of tasks is wrong.**
- **Errors in the use of the resource management (buffer pool)**

A.5 Hardware interface

A.5.1 Description

Issues related to the interface between hardware and software. “Hardware” here refers not to processors, etc, but to the developed, external hardware.

A.5.2 Examples

Name	Wrong code	Corrected code	Description

- **Inconsistencies to actual hardware (e.g. hardware has more pins that the developer thought)**
- **Read/wrote wrong external hardware registers**

Hardware documentation
- **manual is wrong:**
  - **insufficient description (side effects),**
  - **timing constraints in HW**

A.5.3 Distinction to other classes

- All: hardware is always chosen over other classes (see hierarchy)

A.6 Understandability

A.6.1 Description

Issues that are related to understanding the source code, including comments, without functional impact.

A.6.2 Examples

Name	Wrong code	Corrected code	Description

Naming of identifiers	int uselessIdentifier;
	
	const int abc = 0x808;
		int betterIdentifier;
		
		const int C_ABC = 0x808;
			conding standard

Code comments	• clarification and/or additional information regarding the description in the comment • or comment totally missing		
module/method descriptions	• clarification and/or additional information regarding the description in the comment • or comment totally missing		
• **Dead code**			

A.6.3 Distinction to other classes

- Algorithms: anything that doesn't in any way change the functionality, e.g. "x=x;" go to understandability

A.7 Beautification

A.7.1 Description

Issues regarding code formatting and comment style.

A.7.2 Examples

Name	Wrong code	Corrected code	Description
• **typos** • **language grammar problems** • **code/comment alignment (indentation)**			

A.7.3 Distinction to other classes

- typos and language grammar issues that can **NOT** lead to misunderstandings go to beautification; others go into understandability
  Missing plural endings e.g. go to understandability

A.8 Hierarchy of classes

Sometimes it might be that a fault might fall into different categories. Therefore, this hierarchy is provided, which is built up considering RUAG priorities.
Beautification (i.e. Typos, grammar errors) get the highest priority, so that a typo error in a How description (interface) or in a usage constraint description (robustness) is not associated with the high severity that robustness or interface defects might cause.

1. Hardware Interface

2. Timing/Concurrency
3. Interface <-> Algorithm (are orthogonal)
4. Robustness
5. Understandability
6. Beautification

When in doubt, always choose the class that is higher in the hierarchy.

Appendix B UNIT TEST INTERVIEW

B.1 Questions

1. For how many months/years have you done unit testing?
2. Do you start to execute the test cases while implementing or at the end?
3. Do you find defects with unit testing after the first code inspection?
4. What kinds of fault do you find?
5. How long does it usually take to create a unit test script (for one module / for one function)?
6. How often do you execute the unit tests?
7. If a unit test fails, how is the approx. ratio in % between test case wrong and code wrong?
8. How long does it usually take to adapt a test case?
9. How long does it usually take to find and fix one defect in the source code?
10. How would you grade the unit test activity in terms of effectiveness of finding defects on a scale from 1 (Very good) to 5 (Very bad)?

B.2 Interview & shadowing session results

Question	I1	I2	I3	I4	S1	S2	Conclusion /Average
1	1.5 years	1 year	10 years	2 years	2-3 months	2-3 months	2.5 years
2	End Depends on complexity	End Dedicated unit tester	During	End	End	End	Mostly at the end
3	Yes, 2-3 faults / 50 odules	Yes 1 fault / module	No	No	N/A	N/A	Almost no faults
4	See table below	See table below	See table below	See table below			
5	4-8h	4-8h	2h-2w 2-3d avg	1/3 of dev time 2d per 1w module	---	~30 miutes per function	Avg 1-2d / module
6	-During CI -Before delivery -Some-times during Dev.	-On developer request 1-2 faults / module	-During CI -Before delivery 1-2 faults / module	-During CI -Before delivery <<1 fault / module	2 faults found in the module	No faults found	~1 fault / module

	Cannot say how many faults / module						
7	80-90 : 20-10	80:20	95:5	95:5	~80:20	~95:5	90-95% TC wrong
8	30m-60m	15m-60m	5m	5m	~5m	~5m	~15m
9	---	---	5m	5m	8m	---	~5m
10	2	3	3	4.5	---	---	~3.1

Table 29: Summary of interview & shadowing session results

Fault	**Often**	**Sometimes**	**Never**	**Notes**	**Fault class**
Faults described by the participants					
Uninitialized variables	X X				Algorithm
Miscalculated index (in array e.g.)	X				Algorithm
Logical errors	X X				Algorithm
Boundary, ranges	X X	X			Robustness
Other fault types (asked by the interviewees					
wrong conditions	X X	X X		→Often	Algorithm
assignments	X X	X	X	→Often	Algorithm
function calls		X	X X X	→Never	Algorithm
wrong statement order		X X	X X	→Very few	Algorithm
casts		X	X X X	→Never	Robustness
asserts			X X X	→Never	Robustness
usage constraints		X	X X X	→Never	Robustness
hardware problems			X X X X	→Never	Hardware interface
concurrency			X X X X	→Never	Timing/Concurrency
timing			X X X X	→Never	Timing/Concurrency

Table 30: Recognized fault types in UT by interview participants

Summary of important comments per question

2) Do you start to execute the test cases while implementing or at the end?
- If the module is complex, one function is implemented and tested at a time
- UT responsible recommends UT as debugging
- UT is not a very interesting activity

4) What kinds of fault do you find?
- Problems with the simulator
- "UT is more a check that everything has been covered."

7) If a unit test fails, how is the approx. ratio in % between test case wrong and code wrong?
- Script language wasn’t straight forward, i.e. it took some time to learn Cantata
- Developer changes SW while developing in parallel, i.e. the TC is always one step behind. (Dedicated Unit Tester)

8) How long does it usually take to adapt a test case?
- Change structure can lead to an almost complete rewriting of the test case (~ ½ day of work)
- Typical way of work is copy & paste a TC and adapt values.

10) How would you grade the unit test activity in terms of effectiveness of finding defects on a scale from 1 (Very good) to 5 (Very bad)?
-Extra work forces you to work better
-While developing the TCs and looking at the code, one finds errors in the code.
-Depends on the skill of the developer, if the developer is skilled, very few bugs are found.
-Good if used as debugging, worse when using at the end and almost useless in case of dedicated unit tester
-Quite bad ability to find defects
-*The* way to guarantee 100% coverage
-UT is also sometimes used after CI (even less faults)
-Not useful for this type of software (low level, not complex software), more useful maybe for bigger systems
-Wouldn’t use UT for this kind of software if it wasn’t demanded by the customer
-Could find maybe more defects if it was used more like a black box testing approach, but is difficult since one needs to know the inner structure to create the test cases □ Hard to make UT efficient
-Not our best tool to find defects

LIST OF FIGURES

LIST OF TABLES

GLOSSARY

(F-)IT	Formal Integration Test
(F-)VT-TS	Formal Validation Test against the Technical Specification
(MOM-) DEMO	Defect measurement options model
(MOM-) SECMO	Setup & execution cost measurement options model
AB	Activity to which a defect belongs-to, i.e. the VA from which the defect slipped through. Essentially the same as PB.
ADC	Adaptive defect classification
AF	Activity in which the defect was found, i.e. the VA to which the defect slipped to. Essentially the same as PF.
AR	Acceptance Review
CI	Code Inspection
CP	Change proposal
DDR	Detailed Design Review
DR	Design Review
ECSS	European Cooperation for Space Standardization
FST	Fault slip through
FT	Fault type
IG	Improvement gain
I-IT	Informal Integration Test
IP	Improvement potential
ISS	Iterative Selection Strategy
I-VT-TS	Informal Validation Test against the Technical Specification
MOM	Measurement options model
OF	Overlap factor
PB	Phase to which a defect belongs-to, i.e. the VA from which the defect slipped through
PF	Phase in which the defect was found, i.e. the VA to which the defect slipped to
RRC	Removal & regression cost
SEC	Setup & execution cost
TC	Test case
TRR	Test Readiness Review
UT	Unit Test
VA	Verification & Validation Activity
VT-RB	Validation Test against the Requirements Baseline
X-CI	External Code Inspection
X-DDR	External Detailed Design Review
X-DR	External Design Review

X-VT	External Acceptance Review

BIBLIOGRAPHY

[AHM09] E. Ahmad and B. Raza, "Towards optimizing verification and validation activities in space industry," Master's thesis, Blekinge Tekniska Högskola, Ronneby, 2009.

[AND06] B. Andersen and T. Fagerhaug, *Root Cause Analysis: Simplified Tools and Techniques, Second Edition*, 2nd ed. ASQ Quality Press, June 2006.

[AVI00] A. Avižienis, J.-C. Laprie, B. Randell, and Vytautas, "Fundamental concepts of dependability," Tech. Rep. CS-TR-739, October 2000. [Online]. Available: http://eprints.ncl.ac.uk/file_store/trs/739.pdf

[BAS88] V. R. Basili and H. D. Rombach, "The tame project: towards improvement-oriented software environments," *Software Engineering, IEEE Transactions on*, vol. 14, no. 6, pp. 758-773, 1988. [Online]. Available: http://dx.doi.org/10.1109/32.6156

[BAS94] V. Basili, G. Caldiera, and D. H. Rombach, "The experience factory," in *Encyclopedia of Software Engineering*, J. Marciniak, Ed. John Wiley & Sons, 1994. [Online]. Available: https://docweb.lrz-muenchen.de/cgi-bin/doc/nph-webdoc.cgi/000110A/http/scholar.google.de/scholar=3fhl=3dde&#38;lr=3d&#38;cluster=3d4068380033007143449

[BAS94-2] V. Basili, G. Caldiera, and D. H. Rombach, "The goal question metric approach," in *Encyclopedia of Software Engineering*, J. Marciniak, Ed. John Wiley & Sons, 1994

[BRA06] J. S. Bradbury, J. R. Cordy, and J. Dingel, "An empirical framework for comparing effectiveness of testing and property-based formal analysis," *SIGSOFT Softw. Eng. Notes*, vol. 31, no. 1, pp. 2-5, 2006. [Online]. Available: http://dx.doi.org/10.1145/1108768.1108795

[BUG09] Bugzilla, *Defect Tracking System*, developed by the Mozilla Foundation, Version 3.4.1, 2009. Website: www.bugzilla.org.

[CAN09] Cantata++, *Cantata++ for testing C, C++ and Java*, developed by IPL Information Processing Ltd., Version 5, 2007. Website: http://www.iplbath.com/products/tools/pt400.uk.php, 2009, last accessed: 2009-08-15.

[CHI92] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M. Y. Wong, "Orthogonal defect classification-a concept for in-process measurements," *Software Engineering, IEEE Transactions on*, vol. 18, no. 11, pp. 943-956, 1992. [Online]. Available: http://dx.doi.org/10.1109/32.177364

[CLQ09] Rational ClearQuest, *Comprehensive Software Change Management*, developed by IBM, 2009. Website: http://www-01.ibm.com/software/awdtools/clearquest.

[CVS06] CVSTrac, *A Web-Based Bug And Patch-Set Tracking System For CVS, Subversion and GIT*, developed by D. Richard Hipp, Version 2.0.1, 2006. Website: http://cvstrac.org.

[DAM05] L. O. Damm and L. Lundberg, "Identification of test process improvements by combining fault trigger classification and faults-slip-through measurement," in *Empirical Software Engineering, 2005. 2005 International Symposium on*, 2005, pp. 10 pp.+. [Online]. Available: http://dx.doi.org/10.1109/ISESE.2005.1541824

[DAM06] L.-O. Damm, L. Lundberg, and C. Wohlin, "Faults-slip-through - a concept for measuring the efficiency of the test process," *Software Process: Improvement and Practice*, vol. 11, no. 1, pp. 47-59, 2006. [Online]. Available: http://dx.doi.org/10.1002/spip.253

[DEM86] Deming, W. Edwards. *Out of the Crisis*. Cambridge, Mass.: MIT, Center for Advanced Engineering Study, 1986.

[DMA86] T. Demarco, *Controlling Software Projects: Management, Measurement, and Estimates*. Prentice Hall PTR, June 1986.

[ECSE40-1] ECSS Std E-40 Part 1B, Software General Requirements, 2003.

[ECSE40-2] ECSS Std E-40 Part 2B, Software General Requirements, 2005

[ECSQ80] ECSS Std Q-80B, Software Product Assurance, 2003. .

[EMA98] K. El Emam and I. Wieczorek, "The repeatability of code defect classifications," in *Software Reliability Engineering, 1998. Proceedings. The Ninth International Symposium on*, 1998, pp. 322-333. [Online]. Available: http://dx.doi.org/10.1109/ISSRE.1998.730897

[FEL09] R. Feldt, E. Ahmad, B. Raza, E. Hult, and T. Nordebäck, "Evolving the ECSS standards and their Use: Experience based on Industrial Case Studies," in *Data Systems in Aerospace*, 2009.

[FEL09-2] R. Feldt, R. Torkar, J. Schulte, and P. Preissing, "An adaptive defect classification framework for software development," September 2009, in submission.

[FOW90] P. Fowler and Stan Rifkin, "Software Engineering Process Group Guide," Tech. Rep. CMU/SEI-90-TR-024, September 1990. [Online]. Available: http://www.sei.cmu.edu/pub/documents/90.reports/pdf/tr24.90.pdf

[FRE05] B. Freimut, C. Denger, and M. Ketterer, "An industrial case study of implementing and validating defect classification for process improvement and quality management," in *Software Metrics, 2005. 11th IEEE International Symposium*, 2005, pp. 10 pp.+. [Online]. Available: http://dx.doi.org/10.1109/METRICS.2005.10

[GRA92] R. B. Grady, *Practical software metrics for project management and process improvement*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1992.

[IBM02] IBM, “Orthogonal Defect Classification v5.11”, Website: http://www.research.ibm.com/softeng/ODC/DETODC.HTM, 2002, last accessed: 2009-04-20.

[IEE610] IEEE Std 610.12-1990, IEEE standard glossary of software engineering terminology, 1990.

[IEE1044] IEEE Std 1044-1993. IEEE Standard Classification for Software Anomalies, 1993.

[IEE1061] IEEE Std 1061-1998, IEEE standard for a software quality metrics methodology, 1998.

[IPL09] IPL, Cantata++ Technical Brief, 2009. [Online]. Available: http://www.iplbath.com/pdf/p0003.uk.pdf

[JON97] M. Jones, U. K. Mortensen, and J. Fairclough, "The esa software engineering standards: past, present and future," in *Software Engineering Standards Symposium and Forum, 1997. 'Emerging International Standards'. ISESS 97., Third IEEE International*, 1997, pp. 119-126. [Online]. Available: http://dx.doi.org/10.1109/SESS.1997.595952

[KIK01] N. Kikuchi and T. Kikuno, "Improving the testing process by program static analysis," in *Software Engineering Conference, 2001. APSEC 2001. Eighth Asia-Pacific*, 2001, pp. 195-201. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=991477

[KIT98] B. Kitchenham and S. Linkman, "Validation, verification, and testing: diversity rules," *Software, IEEE*, vol. 15, no. 4, pp. 46-49, 1998. [Online]. Available: http://dx.doi.org/10.1109/52.687944

[LET05] T. C. Lethbridge, S. E. Sim, and J. Singer, "Studying software engineers: Data collection techniques for software field studies," *Empirical Software Engineering*, vol. 10, no. 3, pp. 311-341, July 2005.

[LIG08] P. Liggesmeyer, "Lecture on safety and reliability of embedded systems," 2008. [Online]. Available:
http://agde.informatik.uni-kl.de/teaching/suze/ws2008/material/folien/SRES_02_Terms.pdf

[LIT00] B. Littlewood, P. T. Popov, L. Strigini, and N. Shryane, "Modeling the effects of combining diverse software fault detection techniques," *Software Engineering, IEEE Transactions on*, vol. 26, no. 12, pp. 1157-1167, 2000. [Online]. Available: http://dx.doi.org/10.1109/32.888629

[LUT03] R. Lutz and C. Mikulski, "Orthogonal defect classification for projects," NASA Jet Propulsion Laboratory, Tech. Rep., 2003. [Online]. Available: http://hdl.handle.net/2014/7339

[MCA93] D. McAndrews, "Establishing a software measurement process," Software Engineering Institute, Tech. Rep. CMU/SEI-93-TR-16, July 1993.

[NAK06] T. Nakamura, L. Hochstein, and V. R. Basili, "Identifying domain-specific defect classes using inspections and change history," in *ISESE '06: Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*. New York, NY, USA: ACM, 2006, pp. 346-355. [Online]. Available: http://dx.doi.org/10.1145/1159733.1159785

[RAZ09] B. Raza, E. Ahmad, R. Feldt, and T. Nordebäck, "ECSS Standard Compliant Agile Development for Dependable Space Software - an Industrial Case Study," 2008, in submission.

[RIC04] D. F. Rico, *ROI of Software Process Improvement: Metrics for Project Managers and Software Engineers*. J. Ross Publishing, Inc., 2004.

[RUA09] RUAG Holding, "RUAG Aerospace Sweden AB", Website: http://www.space.se, 2009, last accessed: 2009-08-06.

[SAL04] A. Saltelli, S. Tarantola, F. Campolongo, and M. Ratto, *Sensitivity Analysis in Practice – A Guide to Assessing Scientific Models.* John Wiley & Sons, 2004.

[SCH95] K. Schwaber, "Scrum Development Process," in *OOPSLA'95: Workshop on Business Object Design and Implementation*, 1995.

[SHU02] F. Shull, V. Basili, B. Boehm, A. W. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. Zelkowitz, "What we have learned about fighting defects," in *Software Metrics, 2002. Proceedings. Eighth IEEE Symposium on*, 2002, pp. 249-258. [Online]. Available: http://dx.doi.org/10.1109/METRIC.2002.1011343

[SIM09] SIMLAB, *Simulation Environment for Uncertainty and Sensitivity Analysis*, developed by the Joint Research Centre of the European Commission, Version 2.2, 2009.

[SOL04] R. van Solingen, "Measuring the roi of software process improvement," *IEEE Software*, vol. 21, no. 3, pp. 32-38, 2004. [Online]. Available: http://dx.doi.org/10.1109/MS.2004.1293070

[SPL07] Splint, *Annotation-Assisted Lightweight Static Checking*, developed by the Inexpensive Program Analysis Group at the Computer Science Department of University of Virginia, Version 3.1.2, 2007.

[SUB09] Subclipse, *Eclipse Team Provider plug-in providing support for Subversion within the Eclipse IDE*, developed by CollabNet Inc., Version 1.6, 2009. Website: http://subclipse.tigris.org.

[TAY02] C. B. Tayntor, *Six Sigma Software Development*, 1st ed. Auerbach Publications, 2002.

[UOV03] University of Virginia, Splint User's Manual, 2003. [Online]. Available: http://www.splint.org/downloads/manual.pdf

[WAG05] S. Wagner and T. Seifert, "Software quality economics for defect-detection techniques using failure prediction," in *3-WoSQ: Proceedings of the third workshop on Software quality*. New York, NY, USA: ACM, 2005, pp. 1-6. [Online]. Available: http://dx.doi.org/10.1145/1083292.1083296

[WAG06] S. Wagner, "A model and sensitivity analysis of the quality economics of defect-detection techniques," in *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*. New York, NY, USA: ACM, 2006, pp. 73-84. [Online]. Available: http://dx.doi.org/10.1145/1146238.1146247

[WAG08] S. Wagner, "Defect classification and defect types revisited," in *DEFECTS '08: Proceedings of the 2008 workshop on Defects in large software systems*. New York, NY, USA: ACM, 2008, pp. 39-40. [Online]. Available: http://dx.doi.org/10.1145/1390817.1390829

[WOJ07] M. A. Wojcicki and P. Strooper, "An iterative empirical strategy for the systematic selection of a combination of verification and validation technologies," in *Software Quality, 2007. WoSQ'07: ICSE Workshops 2007. Fifth International Workshop on*, 2007, p. 9. [Online]. Available: http://dx.doi.org/http://dx.doi.org/10.1109/WOSQ.2007.4

[WOO97] M. Wood, M. Roper, A. Brooks, and J. Miller, "Comparing and combining software defect detection techniques: a replicated empirical study," in *ESEC '97/FSE-5: Proceedings of the 6th European SOFTWARE ENGINEERING conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering*. New York, NY, USA: Springer-Verlag New York, Inc., 1997, pp. 262-277. [Online]. Available: http://dx.doi.org/10.1145/267895.267915