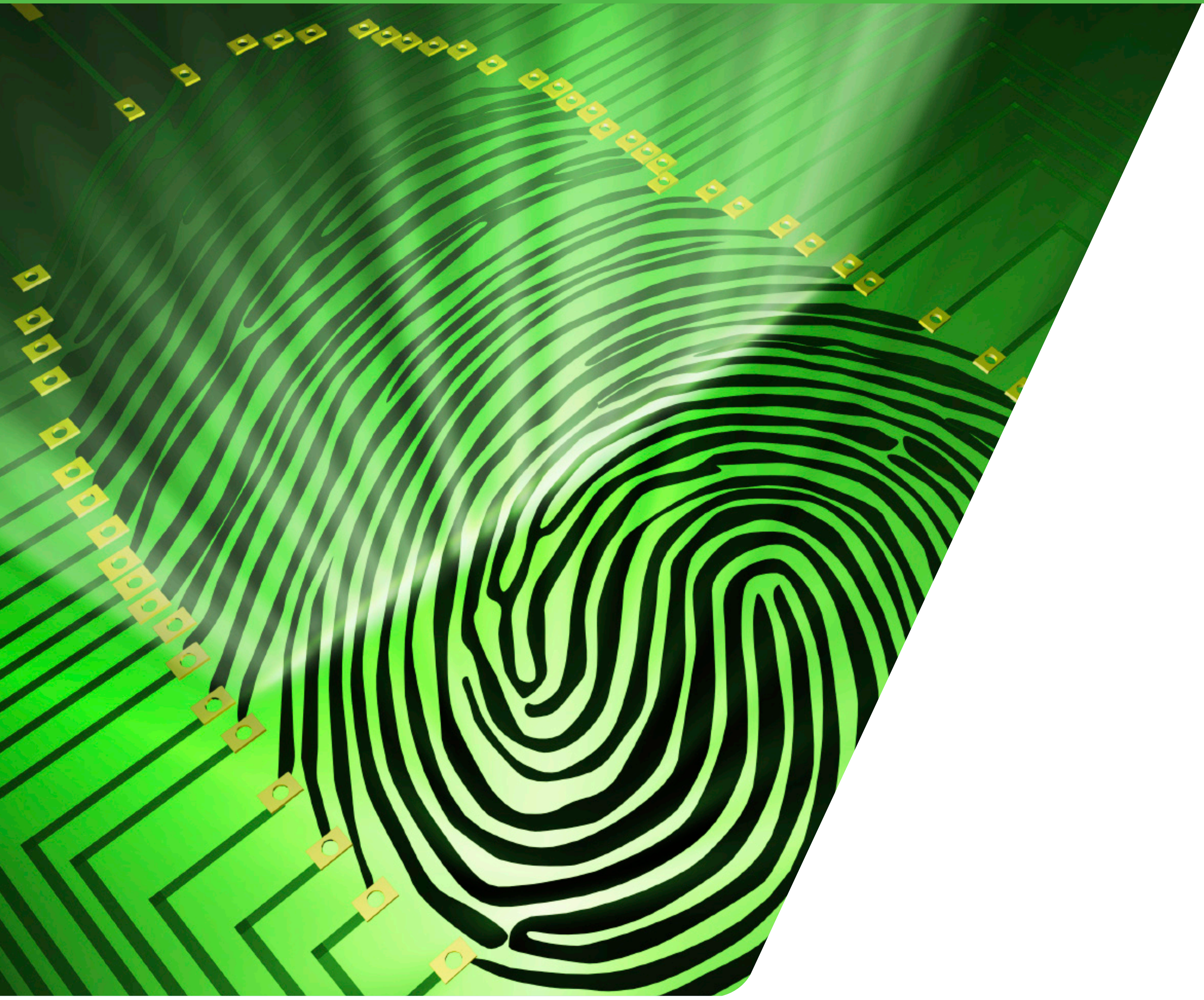
VTT PUBLICATIONS 736



Jukka-Pekka Pesola

Building Framework for Early Product Verification and Validation

VTT PUBLICATIONS 736

Building Framework for Early Product Verification and Validation

Jukka-Pekka Pesola

VTT

ISBN 978-951-38-7393-6 (URL: http://www.vtt.fi/publications/index.jsp)
ISSN 1455-0849 (URL: http://www.vtt.fi/publications/index.jsp)

JULKAISIJA – UTGIVARE – PUBLISHER

VTT, Vuorimiehentie 5, PL 1000, 02044 VTT
puh. vaihde 020 722 111, faksi 020 722 4374

VTT, Bergsmansvägen 5, PB 1000, 02044 VTT
tel. växel 020 722 111, fax 020 722 4374

VTT Technical Research Centre of Finland, Vuorimiehentie 5, P.O. Box 1000, FI-02044 VTT, Finland
phone internat. +358 20 722 111, fax + 358 20 722 4374

Technical editing Mirjami Pullinen

Jukka-Pekka Pesola. Building Framework for Early Product Verification and Validation [Kehikko tuotteen aikaiseen verifiointiin ja validointiin]. Espoo 2010. VTT Publications 736. 75 p.

Keywords validation, verification, testing, V&V, product lifecycle

Abstract

This thesis report presents a step-by-step approach on verifying and validating products early during the development. The final result is a framework that helps detect defects during the early phases of the product lifecycle and prevent them during the later phases.

First, this report presents extensive research on the main verification and validation activities. These activities – simulation, prototyping, reviews and testing – are presented in the report, along with some suggestions for planning them effectively.

The challenges for early product verification and validation were identified and researched by conducting two large studies. The first was a survey that was conducted to gain insight on the current state of the early verification and validation process in the industrial environment. The survey resulted in valuable set of results regarding the current state-of-the-practice of verification and validation in software development in the European setting.

The second research was a literature study on state-of-the-art of early verification and validation processes, methods and techniques. The study resulted in exhaustive collection of methods and techniques that can be applied for product early verification and validation in the different phases of the project. The main results of both of these studies are presented in this thesis.

Based on the results of the studies mentioned above, the requirements for the framework were defined. As a result, 16 requirements for the framework were specified and these are described in this work.

The design and building of the framework according to the requirements then started. The results, including a set of methods and techniques for applying early verification and validation in product development, are presented in this thesis. Finally, the framework was tested in a real-life industrial case and the experiments and results from this case are presented.

Jukka-Pekka Pesola. Building Framework for Early Product Verification and Validation [Kehikko tuotteen aikaiseen verifiointiin ja validointiin]. Espoo 2010. VTT Publications 736. 75 p.

Avainsanat validation, verification, testing, V&V, product lifecycle

Tiivistelmä

Tässä työssä etsitään ratkaisuja tuotteen verifioinnin ja validoinnin aikaistamiseksi. Työssä käydään läpi vaiheittainen lähestymistapa ongelmaan. Työn lopputuloksena määritellään kehikko, joka auttaa löytämään ohjelmistovikoja jo tuotteen elinkaaren aikaisissa vaiheissa ja estämään niiden siirtymistä myöhempiin vaiheisiin.

Työn aluksi esitetään kattava tutkimus tärkeimmistä verifiointiin ja validointiin käytetyistä toiminnoista. Nämä toiminnot – simulointi, protypointi, katselmoinnit ja testaus – esitellään ja ohjeita niiden suunnittelemiseksi annetaan.

Tuotteen aikaisen verifioinnin ja validoinnin haasteita kerättiin ja tutkittiin tekemällä kaksi laajaa tutkimusta. Näistä ensimmäinen oli teollinen kysely, jonka tarkoitus oli antaa näkemystä aikaisesta verifioinnista ja validoinnista teollisessa ympäristössä. Kyselyn tuloksena saatiin hyödyllinen tietopaketti verifioinnin ja validoinnin käytännöistä Euroopan tason ohjelmistokehityksessä.

Toinen tutkimus oli kirjallinen selvitys nykyisistä verifioinnin ja validoinnin prosesseista, metodeista ja tekniikoista. Tutkimuksessa koottiin yhteen laaja kokoelma metodeja ja tekniikoita, joita voidaan käyttää tuotteen aikaiseen verifiointiin ja validointiin projektin eri vaiheissa. Molempien tutkimusten päätulokset on esitelty tässä työssä.

Kehikolle määritettiin vaatimukset yllämainittuihin tutkimustuloksiin perustuen. Vaatimusmäärittelyn tuloksena saatiin kaikkiaan kuusitoista vaatimusta, jotka esitellään tässä työssä.

Tämän jälkeen kehikon suunnittelu ja rakentaminen vaatimusten mukaiseksi alkoi. Tulokseksi saatiin kehikko, joka sisältää metodeja ja tekniikoita aikaiseen verifiointiin ja validointiin tuotekehityksessä. Lopuksi kehikkoa kokeiltiin oikeassa teollisessa tapauksessa. Tapauksesta saatiin tuloksena kokemuksia, jotka raportoidaan tässä työssä.

Preface

This thesis work was done as part of the Evolve project at VTT Technical Research Centre of Finland. Research in the Evolve project concentrates on finding means for improving the verification and validation process in an industrial project setting. More specifically, Evolve searches for iterative and incremental ways to detect and prevent defects during the early stages of the product's lifecycle. Evolve is a European-level ITEA project with partners from five countries. This thesis was done to support the goals of the Evolve project and work on the subject will continue.

I want to thank Päivi Parviainen from VTT Technical Research Centre of Finland, for her guidance throughout this thesis. Her feedback and support was critical to the outcome of this thesis. I would also like to thank Professor Jukka Riekki and Professor Tapio Seppänen from the University of Oulu for supervising and supporting the work. Finally, I want to thank the Evolve project members, especially Hannu Tanner, for their co-operation and contributions during the work.

Oulu, 24th February, 2010

Jukka-Pekka Pesola

Contents

List of symbols

CQ	Conformiq Qtronic, tool for model-based testing
EA	Enterprise Architect, tool for software design
FQR	Formal Qualification Review
ID	Identification
IDE	Integrated Development Environment
JDT	Java Development Tooling, Eclipse toolset for Java development
MBT	Model-Based Testing
MS	Microsoft
NDA	Non-Disclosure Agreement
OCL	Object Constraint Language
QA	Quality Assurance
RGM	Requirements Generation Model
RUP	Rational Unified Process, incremental product lifecycle model
SCRAM	Scenario-based Requirements Engineering, requirements vaildation technique
SotA	State-of-the-Art
SUT	System Under Test
SW	Software
UML	Unified Modelling Language
VDM	Vienna Development Model, requirements validation technique
V&V	Verification and Validation
XML	eXtensible Markup Language

1. Introduction

During the past few decades, the software (SW) sizes in products have been growing rapidly. This growth has driven companies to invest more and more in product quality management. There is continuous demand for faster and more reliable defect detection and prevention systems, and new possibilities and solutions to meet this demand are being developed all the time.

Product verification and validation (V&V) should be regularly present from the very beginning of the development process. Software testing and bug fixing in the late phase of the development is too time-consuming and often leads to budget and schedule overruns. In 1987, Barry Boehm [1] had already noted that the later SW defects are found, the more expensive they are to fix. Companies can achieve remarkable savings if the defects are identified during the early phases of development work. There are also other researches (for example [2] and [3]) that have justified *why* it is important to apply early verification and validation. However, these researches do not explain *how* this could be carried out.

It can be said that the product development is “an exercise in problem solving.” In such problem solving, the main activity involves determining whether the solution is right. V&V is a process entailing procedures, activities, techniques and tools to ensure that the intended problem is solved correctly. [4, 5]

Validation measures the product’s accuracy with respect to the product requirements while *verification* is performed at each phase and between phases to ensure that product development is being done correctly during each phase and sub-phase. [4] Both aspects are essential because a system that meets its specifications is not necessary technically correct and vice-versa. [6] It is sometimes said that validation can be summarised by the question “Are you building the right thing?”, while verification involves asking “Are you building the thing right?” “Building the right thing” refers to the user's needs; while “building it right” checks that the documented development process was followed. [5]

The term V&V is commonly used as a single expression, but this doesn’t mean that the methodology is an “all or nothing process.” [5] In this document, the term V&V is used to refer to the entire process and the reader can incorporate the parts of V&V which best apply to each situation.

This thesis presents the results of research done in the area of V&V. The main focus was to find ways to apply V&V during the early phases of the product’s lifecycle. In the end, the goal is to create an initial framework iteration, which allows industrial companies to regularly apply the V&V activities and methods from the beginning of the development work. After this thesis is completed, further iterations will be developed as the Evolve project continues.

To support the creation process, this thesis presents a literature study on the current state-of-the-art (SotA) of early V&V. This study lists and describes current methods, techniques and processes found in the scientific literature that are potentially applicable in the product's early V&V process.

Another goal of this thesis was to design and conduct a survey among the project's industrial partners. The aim of the survey was to produce extensive information on the current state-of-the-practice of V&V processes and practices in the industrial setting. These results will form a basis for the V&V framework(s), to make it better suited to the industrial projects' needs.

The next goal is to find and define the requirements for the framework developed in this work and also for the Evolve framework developed during later stages of the project. The results of both the industrial survey and the literature study are used in the definition process.

Finally, the framework defined in this work will be tried out in a real-life industrial case. The purpose of this case is to test the usage and effectiveness of the framework in early defect detection and prevention in a real-life setting.

In this thesis, the issues presented above were approached in a structural manner. This approach starts by introducing some common product lifecycle models as well as the main V&V activities. Afterwards, some typical ways of planning these activities during the product development are presented. This background study, presented in chapter 2, formed the theoretical base for the rest of this thesis, and may be reviewed while reading the later chapters. The next step involved defining the scope of the issues in the context of early V&V. To begin this process, two large studies were conducted. Chapter 3 presents the main results from both of these studies and clarifies their relevance to for this thesis. Based on the findings from the previous sections, chapter 4 defines and presents the requirements for the whole Evolve framework, and also distinguishes those whose implementation is planned during this work. After the requirements were defined, the current version of the framework was drafted. Chapter 5 presents the conceptual-level model of that framework, and introduces and justifies the methods and tools that will be tested in this thesis. The testing is done in an industrial case study and the test results are presented in chapter 6. Chapter 7 discusses the results and findings of this thesis and suggests some other areas that could be developed further. Finally, chapter 8 gives a summary of the main results of this thesis and concludes the work. A list of references can be found in chapter 9.

2. Verification and validation activities

To be able to do effective verification and validation (V&V), the development team needs to select a compatible set of techniques and tools to be used in V&V. The selection is based on the V&V goals of the project. These goals are defined by evaluating the characteristics of development problems and the constraints of the solution. From the end user's perspective, the techniques and tools must ensure a functional, efficient and reliable product within the given time limits. The developer is concerned with these issues as well as with the product's integrity and composition. These concerns provide the guidelines for selecting the relevant V&V activities. [5]

Optimally, V&V activities should be present at every stage of the product lifecycle. In this chapter, some typical product lifecycle models are introduced in order to provide a context for V&V activities. During the product lifecycle, four V&V activities are commonly used: prototyping, simulation, review and testing. In this chapter, a section is dedicated to each of these activities. Finally, at the end of this chapter, the management and planning of the V&V process is studied and some methods for conducting test cases are presented.

2.1 Introduction to product lifecycle models

Product verification and validation is placed on top of the lifecycle model that is used in the development process. Depending on the model, some V&V activities may be present during the different phases of the development. In this section, two commonly used lifecycle models are presented: the V-model and the Rational Unified Process (RUP). There are various other models as well but these two were selected as sample models. The V-model was selected because it is a traditional life-cycle model, in which the V&V viewpoint is emphasised. Rational Unified Process, on the other hand, is commonly used in incremental product development.

2.1.1 The V-model for product verification and validation

The V-model is a software development process in which the process steps are shifted upwards after the coding phase, to form the typical V shape. The V-Model can be presumed to be the extension of the traditional waterfall model in which the steps are placed on a linear graph. The V-Model points out the relationship between each phase of the development lifecycle and shows its associated testing phase.

The development process that applies the V-model follows a structured method in which each phase can be implemented after the detailed documentation of the previous phase. Like test designing, testing activities start at the beginning of the project, well before the actual implementation; therefore, the total project time can be reduced. [7]

Figure 1 shows a V-model that graphically describes the systems engineering approach. [8] The core project development processes are shown inside the dashed lines in the middle of the model. The V-diagram also contains two wings that show a more extensive project lifecycle from the initial project identification through system retirement or replacement at the end of the lifecycle. [8]

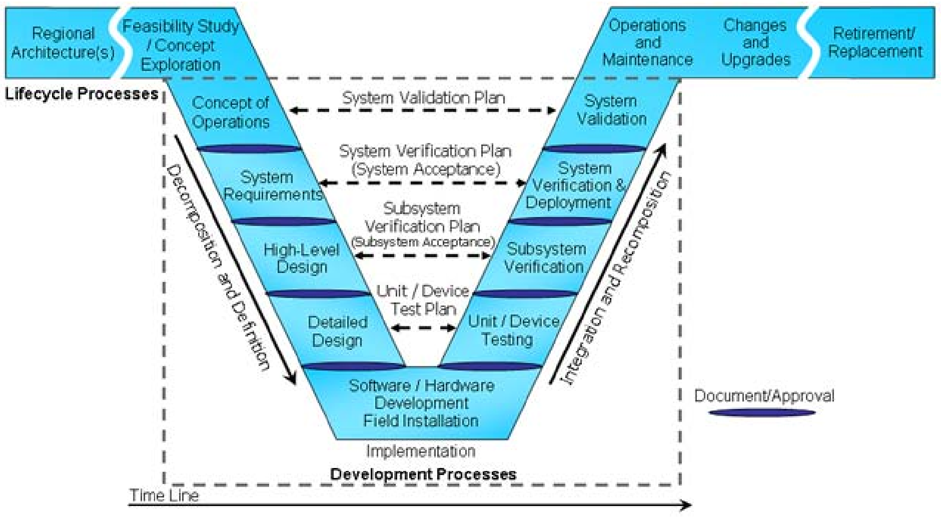


Figure 1. V-model for the product development process.

During the decomposition and definition process, each phase produces different verification and validation plans. Later, during the integration and recomposition phases, the system is compared with these plans for verification and validation. System validation plans are created during the concept of operations phase and system verification plans are put together during the definition of system requirements and the high-level design phases. If there are failures in the integration and recomposition process tests, the project execution returns to the corresponding level of the decomposition and definition i.e. to the phase where the plan for the failed test was initially written.

In the software development process, the V-model is divided into a development and testing phases, as depicted in Figure 2. [9] As in the systems engineering V-Model, the first phases of the process are development phases and the following are testing phases. In each development phase, the corresponding testing plans and specifications are produced. For example, the test plans for acceptance testing are created during the requirements elicitation phase and integration testing specifications are planned during architecture design. Once the test results are available, the results are compared to the documents created during the development phase. The verification is thus done at each testing phase by comparing the system to the specifications that were created during development phases. The validation is done during the system and acceptance testing, in which the system is compared to the user's requirements. [9]

Defects that are found during the different phases of testing can be categorised based on the V-model. The defects that are found during module testing are usually programming faults, which are easy to fix. The faults that appear during integration testing are design errors and reflect changes to the architecture design. The faults that are found in the later test stages can, in the worst case, require that the whole process begin all over again. One example of this kind of fault could be that customer requirements have been misunderstood and the requirements for the system need to be changed; this can be very costly. The later the defects are identified in this model, the greater the costs of solving them. [9]

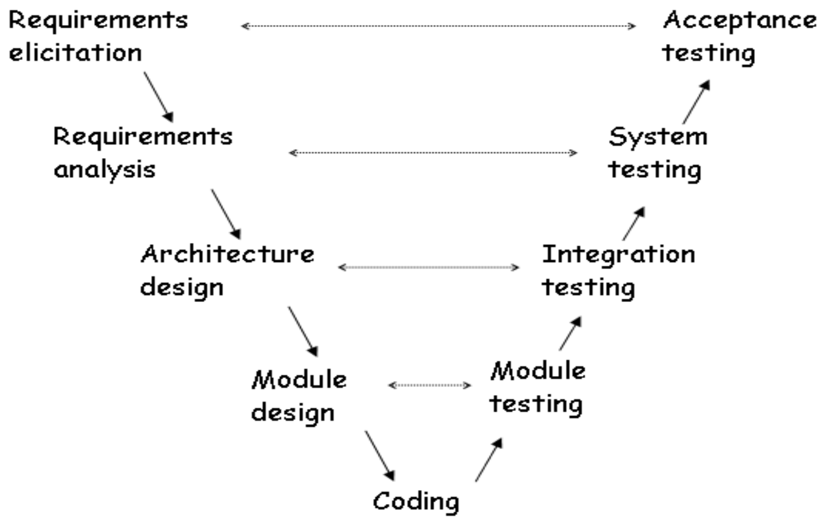


Figure 2. V-model for the software development process.

2.1.2 The IBM Rational Unified Process

The Rational Unified Process (RUP) is a framework for iterative software development process created by the Rational Software Corporation (a division of IBM since 2003). RUP is not a concrete process model, but rather an adaptable framework, intended to be tailored by the development organisations and software project teams that select the elements of the process best suited to their needs. [10] RUP suggests a process flow that is iterative and incremental, providing the evolutionary feel that is essential in modern software development. [11]

The RUP specifies a project lifecycle consisting of four phases (see Figure 3). [12] These time wise separated phases allow the process to be presented at a high level in a similar sequential way as the waterfall model. The key to the process lies in the iterations of development that fall within all of the phases. In addition, each phase has one key objective and milestone that denotes that the phase is complete. [10, 12]

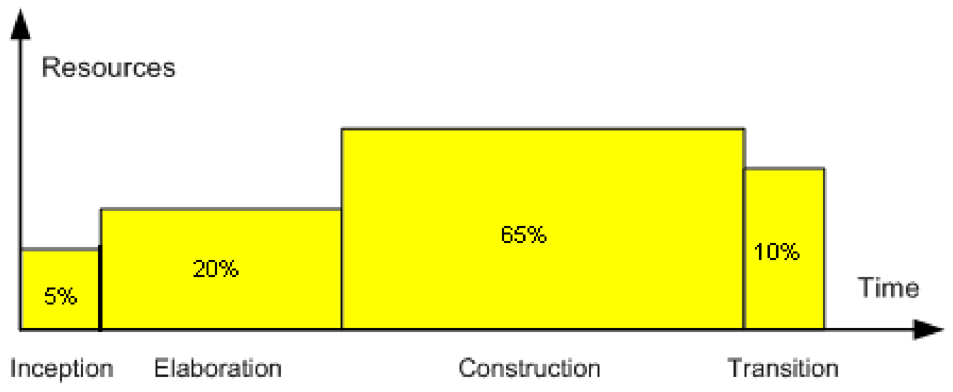


Figure 3. Workload between phases on the RUP project.

1. *Inception phase*

The primary goal of this phase is to establish the business case and launch the project. In this phase, only a small percentage of use cases is selected to support the initial business case. This work continues later in the elaboration phase. The activities during this case include, for example

- defining the scope of the proposed system; beginning to identify the possible interfaces to related systems
- outlining a possible architecture for the system to check its feasibility for the next phase
- building a proof-of-concept prototype to demonstrate the system to potential users or customers.

These efforts are continued to the point at which it appears economically worthwhile to develop the product. The intent is to minimise the costs until it is found that the system is feasible. In the case of a new system in a new domain, this determination may take considerable time and can lead to several iterations in that event. On the other hand, when an existing system is extended to a new release, this phase can be completed in a few days. [12]

2. *Elaboration phase*

During the elaboration, the goal is to build a stable architecture for the system. A study of the system is also performed to plan the construction phase. To achieve these goals, the tasks of this phase include

- creating an architectural baseline that covers the significant functionality of the system and features important to the stakeholders
- identifying significant risks concerning plans, costs and schedules of later phases
- specifying the levels to be attained in terms of quality attributes such as reliability and response times
- preparing a bid that covers the schedule, the staff needed, and the cost within the limits set by business practices.

3. *Construction phase*

The objective of the construction phase is indicated by its major milestone: the initial operational capability, which signifies a product ready for beta testing. The greatest effort is needed in this project phase and it generally involves more iterations than in the earlier phases. The activities of the construction phase include:

- identifying, describing and carrying out the use case and then extending it to the entire body of use cases
- finishing the analysis, design, implementation and testing
- maintaining the integrity of the architecture and modifying it as needed
- monitoring critical and significant risks carried over from the first two phases.

4. *Transition phase*

The transition phase often begins with the beta release, so the development organisation distributes the operating software product to a representative sample of the actual users. Transition activities include

- preparation activities, such as site preparation
- customer support for the beta tests
- preparing manuals and other documentation for product release
- adjusting the software to operate under the actual parameters of the user environment
- correcting defects found after feedback from the beta tests.

Once all objectives are met, the Product Release Milestone is reached and the development cycle ends. [12]

As stated, the RUP process framework is designed for the iterative development. One notable aspect of the framework is a division of the workload between different tasks during each phase of the project. Figure 4 provides a diagram illustrating how the relative emphasis of different disciplines changes over the course of the project. [10] It can be seen that all activities take place during each phase, but the workload for the activities varies depending on the ongoing phase. For example, requirements management lasts throughout the project life cycle but the greatest effort is required during the inception and elaboration phases. Testing, on the other hand, starts during the inception phase with a few minor activities but it is then divided quite evenly amongst the three following stages.

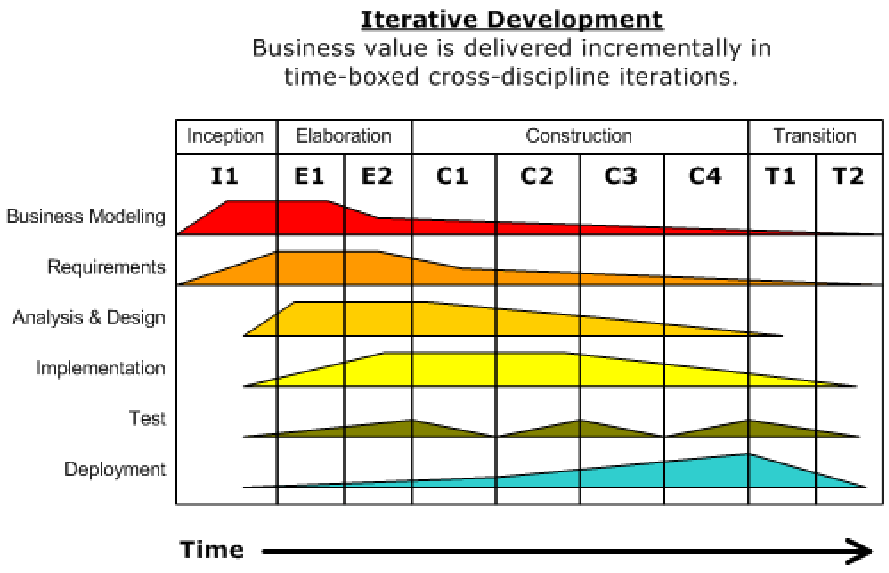


Figure 4. Activities in iterative development process.

2.2 Prototyping

Prototyping is the process of building a model of a system. This model, the prototype, is often used as part of the product design process to make it available for engineers and designers to explore design alternatives, test theories and confirm performance prior to starting the production of a new product. Developers aim to tailor the prototype according to the specific unknowns still present in the intended design. For example, prototypes can be used to confirm and verify a customer's interest in a proposed design. Prototypes can also be used to check the performance or ensure the suitability of a specific design approach. [13, 14] According to Pressman [11], prototyping is an iterative and evolutionary process that can be used as a stand-alone process model; more commonly, however, it is a technique that can be implemented within the context of some other process model.

The advantages of prototyping are that [14]

- it reduces development time and costs
- it requires user involvement
- developers receive user feedback from the early phase of the development
- system implementation is facilitated since users know what to expect
- it results in higher user satisfaction
- it improves developers' possibilities to enhance the system in the future.

The disadvantages of prototyping are that [14]

- it can lead to insufficient analysis
- users may expect the performance of the end system to be the same as the prototype
- developers can become too attached to the prototypes
- it can cause systems to be implemented before they are ready and left unfinished

- it can lead to incomplete documentation
- if some sophisticated prototypes (4th GL or CASE Tools) are used, the time saving benefit of prototyping can be lost.

In most cases, prototypes increase the quality and the amount of communication between the developer/analyst and the end user. As a result, it has been taken into use in a greater number of projects of different types. In the early 1980s, prototyping was used by organisations approximately 30% of the time in development projects. By the early 1990s, its use had increased to 60% [13].

In [13] there is a collection of prototyping guidelines that specify when to use software prototyping. The authors have conducted the study to find out how many of these popular guidelines that appear in information system literature were actually used by organisations with developed prototypes. The question as to whether or not compliance affected the system’s success (measured by the user's stated level of satisfaction) was also evaluated.

Based on the results of their research, the authors found that the industry followed only six of the seventeen guidelines recommended in the literature. These six guidelines were the only ones whose adherence was found to have a statistical effect on the system’s success [13, 14]:

- Prototyping should be employed only when users are actively able to participate in project development.
- Prototyping shouldn’t be used if users lack experience with prototyping.
- Developers should have prototyping experience or receive training on prototyping.
- Evolutionary prototyping should only be used if the developers are given access to prototyping support tools.
- Prototyping can be successfully used if experimentation and learning are needed before there can be full commitment to a project.
- Prototyping is not necessary if the developer has extensive experience with the application domain.

There are many different techniques and approaches for prototyping, but usually they fall into one of two main categories: throw-away prototyping and evolutionary prototyping. The main difference between these two categories is that in throw-away prototyping, a prototype is created to help inspect and analyse system characteristics, but it will eventually be discarded. In evolutionary prototyping, a system prototype with limited functionality is made available to the users early in the development process. This prototype is later modified and extended to produce the final system. [15, 16]

Different prototyping techniques include paper prototyping, Wizard of Oz prototyping and storyboard prototyping. In paper prototyping, screen shots and/or hand-sketched drafts of e.g. the windows, menus or dialog boxes of the system are made, and then tested for usability. This technique can be employed for usability testing of websites, web applications and conventional software. In Wizard of Oz prototyping, the user interacts with the computer system under study which is actually operated by a (hidden) developer – referred to as the “wizard”. The wizard processes input from a user and simulates the output. This form of prototyping is beneficial early on in the design cycle as it provides information about the user’s expectations. In storyboard prototyping, the idea is to use sequences of computer-generated displays, called storyboards, to simulate the functions of the formally implemented system beforehand. This supports the communication of system functions to the user, and makes the trade-offs from non-functional to functional requirements visible, traceable and analysable. Hence the method is suitable for use in the requirements analysis and negotiation phases. [15]

2.3 Simulation

Simulation is defined in [17] as the "process of designing a model of a real or imagined system and conducting experiments with that model". The purpose of simulation is to understand the behaviour of the system or evaluate strategies for the operation of the system. For example, a business may be interested in building a new factory to replace an old one, and want to know whether the increased productivity will justify the investment. A simulation could be used to evaluate a model of the new factory. In this case, a simulation can be extremely valuable, since it is very expensive to set up an entire factory to determine its best configuration. [17]

In software engineering, simulations can also provide valuable information. They are used in the design phase to explore alternatives, spot flaws, and optimise product performance before the detailed design. In verification, simulations provide many new tools to optimise testing, system profiling and test material generation. The partial implementations can also be tested with the help of the simulation. The results of the simulation can be used while making important decisions concerning the functionality and allocation during the early phases of a product lifecycle. The usefulness of a simulation doesn't end in the design phase; optimising existing systems and researching the effects of possible modifications are other typical use cases for simulations. [18]

The benefits of simulation/animation are [18, 19, 20]:

Validation of a concrete object (in the context of requirements engineering). Simulations (as well as prototyping) have the major benefit of validating a concrete and executable object (the model of the system). This can improve the quality of a requirements specification since users are not relying only on inspections and reviews of textual requirements specifications.

No need to write code. In contrast to classical prototyping, almost no code needs to be written in order to run simulations. Depending on the simulation tool used, some code fragments may be needed in order to implement specific execution details.

No need to create application-specific GUIs. Simulation tools can usually directly animate the graphical notation of the supported modelling language(s). As long as all parties working with simulation are familiar with the animated modelling languages, there is no (immediate) need for creating application-specific GUIs in order to interact with the executed models.

White-box visualisation. The animation of modelling languages provides a means of visualising the internal behaviour of the system model. That way, it is possible to validate the internal mechanisms of a system model by inspection. In contrast, classical prototyping usually relies only on application-specific GUIs for interacting with the prototype (black-box view).

Better understanding of the system. Simulations help to better understand the behaviour of the future system. Sometimes the stakeholders are not sure of exactly what they want or how to voice their ideas and desires. Additionally, a system that is well understood allows costs to be predicted more accurately.

Analytical evaluation. For the most complex systems, it is impossible to build a mathematical model that can be evaluated analytically. Simulations are thus the only way to explore these systems.

Neglected random factors. Simulations provide better control of testing conditions and variables, when compared to testing with real systems.

Time scale is adjustable. Lengthy simulations can be done in much less time and in addition, more detail can be achieved.

Flexibility. Simulations enable different system designs to be compared without implementing several prototypes.

The drawbacks of simulation/animation are [18, 19, 20]:

No way to validate discrete systems exhaustively. Like testing, simulations have major limitations and drawbacks. Complex systems with a large number of states cannot be tested or simulated exhaustively.

Model animation is not always stakeholder-friendly. The model animation highlights the current state of the model, which in turn is represented by the notations modelling languages. Non-technical stakeholders might experience difficulties in reading and interpreting such animations.

Low abstraction level. Simulation (as well as prototyping) often requires additional implementation details of the future system in order to be executable at all. This leads to the problem of over-specification (the model is too solution/implementation-oriented).

In requirements engineering, semantically correct and formal models are required. As long as the requirements engineering process is ongoing, the requirements engineers have to deal with incomplete and informally stated requirements. In this case, requirements engineers need to define a semantically correct and formal system model for running simulations

Need to model the system's environment. Embedded systems are part of larger systems and involve complex interaction with their environment. For example, a prototype would probably allow connection to the existing environment, but in simulations, the environment also has to be simulated.

Non-functional requirements are not fully supported. Simulation focuses on the behavioural aspects of a system. However, non-functional requirements, such as reliability, cannot be simulated adequately from requirements.

The use of simulation, animation or emulation has become quite common. According to [17], simulation is used in nearly every engineering, scientific, and technological discipline. The simulation techniques are employed in the design of new systems and in the analysis of existing ones.

2.4 Reviews

Reviews involve evaluating software elements or a project status to assess any deviations from planned results and to make suggestions for improvement. This evaluation is based on a formal process such as the management review process, the technical review process, the software inspection process or the walkthrough process. [21]

Reviews can be used to support objectives associated with software quality assurance, project management, and configuration management, or similar control functions, valid throughout software development and maintenance. The following figure (Figure 5) presents the relation of some quality assurance processes to products and projects. [22]

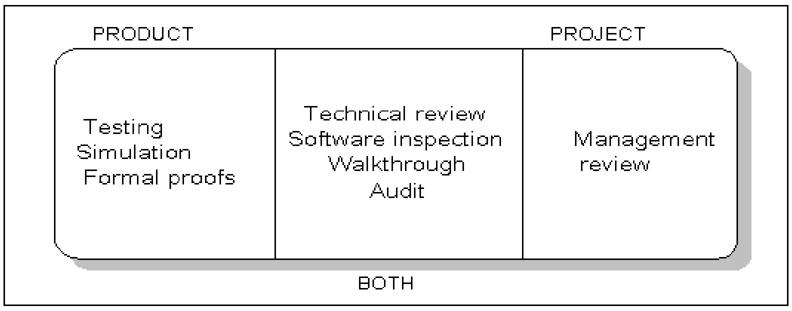


Figure 5. Quality assurance (QA) processes in a project.

In this work, the review methods are placed into one of the following four categories characterised by the strategy that drives the review process: [23]

1. *Management review.* A formal evaluation of a project level plan or project status of the plan by a designated review team. [22]
2. *Technical reviews.* In technical review (or formal review), the author of the work product introduces it to the rest of the reviewers. The flow of the review depends on the presentation and issues brought up by the reviewers.
3. *Inspections.* In an inspection, a list of criteria that the software must satisfy is drafted and software elements are checked to ensure that they fulfil the criteria. [23]
4. *Walk-throughs*. Walk-throughs are used to examine source code as opposed to design and requirements documents. The author of the code is usually present to answer participants' questions.

The main differences between these categories lie in the purpose of the review. A management review is done in order to provide recommendations for progressing on project activities according to plan, to change the project's direction or to maintain global control of the project. A technical review is done in order to evaluate a specific software element and show management that the element conforms to its specifications or that changes to the element are properly implemented. An inspection is done in order to find defects in the software element and check to ensure it conforms to applicable standards. A walk-through is done in order to identify defects, omissions and contradictions and in order to consider alternative implementations. [23]

In the following chapters, three of these main review categories are described in more detail and some other techniques are briefly presented. Management review is not discussed here since that doesn't fall within the scope of the work.

2.4.1 Technical reviews

A technical review involves a team of qualified personnel who evaluate a software product to determine its suitability for its intended use and identify discrepancies from specifications and standards. It provides management with evidence to confirm whether [22]:

- the software element(s) conform to specifications
- the development (or maintenance) of the software element(s) is being done according to plans, standards, and guidelines applicable for the project
- changes to the software element(s) are properly implemented, and affect only the system areas identified by the change specification.

Technical reviews may also provide the recommendation and examination of various alternatives for implementation. The examination doesn't need to address all aspects of the product. [21]

2.4.2 Inspections

The purpose of the software inspection is to detect and identify software element defects. This is a rigorous, formal peer examination that does one or more of the following [21]:

- verifies that the software product meets its specifications
- verifies that the software product has the specified quality attributes
- verifies that the software product conforms to applicable regulations, standards, guidelines, plans, and procedures
- identifies deviations from standards and specifications
- collects software engineering data (for example, anomaly and effort data)
- provides the collected software engineering data that may be used to improve the inspection process itself and its supporting documentation (for example, checklists)
- uses the data as input to project management decisions as appropriate (e.g. to make trade-offs between additional inspections and additional testing).

Inspections consist of three to six participants. An inspection is led by an impartial facilitator who is trained in inspection techniques. Determining remedial or investigative action for an anomaly is one mandatory task of a software inspection, although the solution should not be implemented at the inspection meeting. [21]

When defects are found in an inspection, they are usually recorded in great detail. Besides the general location of the error in the code, they may include details such as severity, type (e.g. algorithm, documentation, data-usage), and phase-injection (e.g. developer error, design oversight, requirements mistake). Typically this information is saved in a database so the defect metrics can be later analysed and possibly compared to similar metrics from QA. [24]

2.4.3 Walk-throughs

During the walk-through meeting, the author provides an overview of the software element(s) under review. This is followed by a discussion with the participants in which the presenter "walks through" the software element in detail. The progress, errors, suggested changes and improvements mentioned during the walk-through are noted. When the walk-through is finished, the notes are consolidated into one report, which is delivered to the author and other appropriate personnel. [22]

The purpose of walk-through is to evaluate a software product. A walk-through may be held for the purpose of educating an audience regarding a software product. The main objectives are to [21]

- find anomalies
- improve the software product
- consider alternative implementations
- evaluate conformance to standards and specifications
- evaluate the usability and accessibility of the software product.

Other important objectives of the walk-through are e.g. the exchange of techniques, styles and experience of the participants. During a walk-through, several deficiencies may be pointed out (for example, efficiency and readability problems in the software product, modularity problems in design or code, or specifications that cannot be tested). [21]

2.4.4 Other review types

The **over-the-shoulder review** is the most common and informal of code reviews. [24] As the name suggests, a reviewer stands at the author’s workstation while the author walks through a set of code changes. If the reviewer sees something amiss, he/she can engage in a little “spot pair-programming” as the author drafts the solution under the reviewer’s supervision. [24]

Pair-programming can be associated with agile development, but it is also a development process that incorporates continuous code review. Pair-programming involves two developers writing code at the same workstation with only one developer typing at a time and continuous free-form discussion and review. [24]

A **formal qualification review (FQR)** is the test, inspection, or analytical process in which the group of configuration items that comprises a system is checked to ensure that it meets specific contractual performance requirements. [22]

Unlike a single large review, the **active design reviews** approach entails several brief reviews, each of which focuses on part of the work product. Participants are guided by a series of questions posed by the author of the design to encourage a thorough review. [23]

2.5 Software testing

Software is tested in order to find errors that were made inadvertently during the design and construction. [11] Software testing provides stakeholders with information about the quality of the product or service being tested. Software testing techniques include the process of executing a program or application with the intent of finding software bugs. [25]

Software testing can also be defined as the process of validating and verifying that software [25]

- meets the business and technical requirements that guided its design and development
- works as expected
- can be implemented with the same characteristics.

In this section, the process of testing software is studied and a gradual approach to it is presented. The first sub-section presents methods that are used when designing test cases. After that, the levels of the testing are identified. Finally some testing approaches for running tests during the later stages of the testing process are introduced.

2.5.1 Testing methods

There are two basic methods to identify test cases: black-box testing and white-box testing. These two design strategies are used to present the options a test engineer has when designing test cases. In this sub-section, these two strategies are presented briefly. In addition to white box and black box testing, there is a third approach: hybrid testing. Hybrid testing, also known as grey-box testing, combines white and black box testing, but since it is not so common, it was left out of this thesis. [26]

Black-box test design treats the system as a "black-box" i.e. it does not use the knowledge of the internal structure. Black-box test design is usually focused on testing the functional requirements of features. [27] Black-box testing (also known as behavioural testing) involves detailed knowledge of the application domain and the mission the system under test serves. [27] Black-box testing is not an alternative to white box techniques. It is a complementary approach that is likely to uncover different types of errors than the white box approaches. [28]

Black-box testing tries to identify errors in the following categories: (1) incorrect or missing functions, (2) interface errors, (3) errors in data structures or external database access, (4) performance errors, and (5) initialisation and termination errors. [28]

Unlike black-box testing, **white-box test design** allows one to peek inside the "box". It focuses specifically on using internal knowledge of the software to guide the selection of test data. White-box tests find bugs in low-level operations. These structural tests are based on how a system operates. [27]

White-box testing (also known as structural testing) involves detailed understanding of the system. In software development, the testers develop most white-box tests by looking at the code and the data structures themselves. For the test staff, white-box testing can be difficult, because it requires knowledge of low-level details about the system under test. [27]

Both white-box and black-box testing are methods that are used during the test design phase. Either of these two test design methods can be used at any level (see the next sub-section) of testing.

2.5.2 Testing levels

Testing differs depending on the maturity of the software being tested. In a typical software system, the testing process can be divided in three distinct levels: unit/component testing, integration testing and system testing. In this sub-section, these levels and the objective of testing during each level are presented.

Unit testing is that which is done to find out if the unit being tested satisfies its functional specification or if its implemented structure matches the intended design structure. A **unit** is usually the work of one programmer and it consists of several hundred lines of source code or less. [29] The complexity of tests and the errors these tests uncover is limited by the constrained scope established for unit testing. The unit test focuses on the internal processing logic and data structures. [11]

Component testing has the same focus as the unit testing but in component testing the test target is the **component**, which is an integrated aggregate of one or more units. [29]

Even though the units (or components) are individually satisfactory (i.e. the unit and component testing passes successfully), the combination of components can be incorrect or inconsistent. In order to eliminate these kinds of faults, **Integration testing** is done. [29] The problems of putting the units together arise from the interfaces. Integration tests can be used to uncover errors associated with interfacing. The objective is to take tested units and build a program structure that has been dictated by design. [11]

System testing is aimed at revealing bugs that cannot be attributed to components or the combination of single components. System testing concerns issues and behaviours that can only be exposed by testing the entire integrated system or a major part of it. These tests fall outside the scope of the software process and are not conducted solely by software engineers. System testing can include testing for performance, security, accountability, configuration sensitivity, start-up and recovery. [11, 29] Some of these test approaches are introduced in the following sub-section.

2.5.3 Testing approaches

At the different testing levels, there are some specific approaches for testing the software from different perspectives. For system level testing, it is said in [11] that system testing actually entails a series of different tests whose primary purpose is to fully exercise the system. Sometimes the testing approaches are divided into positive and negative ways of testing. In positive testing, the system's correct operation is tested under normal circumstances (e.g. it is used correctly). In negative testing, the system is put under abnormal stress (e.g. false inputs are fed into the system), and the test evaluates how the system manages to execute. In the following, the types of system and integration level tests that are worthwhile for software-based systems are presented.

Regression testing: Each time a new module is added to the integration testing, the software changes. These changes may cause problems with functions that previously worked. Regression testing is the re-execution of a selected subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects. [11]

Smoke testing is designed as a pacing mechanism for time-critical projects, allowing the software team to assess the project frequently. The essential activities encompassed during smoke testing are as follows: [11]

1. Software components are integrated into a build that includes all data files etc. required for product functions.
2. A series of tests is designed to expose errors in the build. The intent should be to uncover the most critical “show-stopping” errors.
3. The build is integrated with other builds, and the entire product is smoke tested daily in its current form.

Recovery testing is a system test that forces the software to fail in a variety of ways and verifies that recovery is executed properly. The objective of this type of testing is to ensure that the computer-based system recovers from faults and resumes processing with little or no downtime. In some cases, processing faults must not cause overall system function to cease. [11]

Stress testing executes a system in a manner that demands resources in abnormal quantity, frequency or volume. Stress tests could be designed, for example, to generate extra interruptions to the program or to use the maximum amount of memory. In essence, the stress testing could be thought of as a process that attempts to "stress the system until it fails." [11] This kind of system stress often uncovers new hidden defects.

Performance testing: For real-time and embedded systems, software must provide more than the required function. It must also fulfil the performance requirements. Performance testing is designed to test the run-time performance of software in a system. The performance tests must be run at the system level to ascertain the actual performance of the system. [11]

Acceptance testing: In **a**cceptance or user acceptance testing, the aim is to demonstrate that the system meets the established requirements. This approach of testing is common in contractual situations, when the passing of acceptance tests obliges a buyer to accept a system. Acceptance testing involves live (or near live) data and environments, and typical user scenarios (not extreme conditions). [27]

2.5.4 Types of testing tools

Many tools are designed to help in the software testing process. In this section, these tools are categorised based on their area of usage and these categories are briefly introduced. The basis for the categorisation is adapted from [30].

Test runners and test management tools enable functional testing through a user interface such as a native graphical user interface, web interface or non-graphical user interface. The test management tools help organise and execute groups of test cases, also called test suites. Test runners are used for test case execution, either automated, manual or both depending on a tool.

Load and performance tools are tools that stress systems with a heavy load on its inputs (especially client-server systems). These tools are often integrated into test runners and are used while doing stress testing for the product.

Static analysis tools are used to analyse programs offline, without executing software. Analysis is usually performed on some version of the source code. Software metrics tools are the typical tools in this category.

Test coverage tools are designed for evaluating the quality of the tests. These tools are sometimes called test evaluation tools.

Test design Tools help to determine what tests need to be run. These tools can also generate test data from some design models.

Test implementation tools assist with testing at runtime. There are, for example, memory leak checkers, comparators, and a wide variety of other tools.

2.6 Verification and validation planning

In product quality management, it is essential to plan V&V work carefully. This planning has to begin before the actual development work begins. Verification and validation of a larger software project involves a lot more than just testing. According to Pressman [11]: "You can't test quality. If it's not there before you begin testing, it won't be there when you're finished testing." This means that quality is incorporated into software throughout the process of verification and validation for software engineering. This quality is then confirmed during the testing.

In this section, the process of planning V&V and testing activities is studied. This is done by presenting some key issues to be considered during the planning of V&V. We also present some commonly used approaches which can be helpful when planning test cases.

2.6.1 Software testing strategy

A strategy for software testing integrates software test case design techniques into a well-planned set of steps that encompass the production of software. A software test strategy provides a road map for the software developer; it defines the moment for planning and undertaking these steps and details how much effort, time, and resources will be required. Any testing strategy must include test planning, test case design, test execution, and the resulting data collection and evaluation. On the other hand, software test strategy should be flexible enough to promote customisation and be rigid enough to encourage reasonable planning and management tracking as the project progresses. [11, 28]

The ideal method i.e. testing the entire project throughout and ensuring it is watertight, is not feasible. This means that it is necessary to make selections and prioritise. According to [31]: “Testing has always been a cost-avoidance activity. We test because it is cheaper to test than not to test.” This statement alludes to the fact that costs incurred in testing are lower than the costs of fixing and repairing faults encountered later in the field. One interesting question to think of is: “When do the total quality costs reach the minimum?” i.e. the moment where the sum of testing costs and failure costs are minimised. [31]

Testing requires a tight focus. It is easy to try to do too much. There is a virtually infinite number of tests that could be run against any nontrivial piece of software or hardware. [27] A software testing strategy must be focused on striking a balance between budget, time and quality.

2.6.2 V&V planning

The Standard for Software Verification and Validation [4] suggests that the planning of the V&V process should rely on the software criticality levels. According to the standard software, the criticality level can be defined based on its intended use and the application of the system in critical or non-critical uses. Some software systems affect critical, life-sustaining systems, while other software systems are non-critical, standalone research tools. The standard uses a software integrity level approach to quantify software criticality.

To plan the V&V processes, software integrity levels should be assigned early in the development process, that is, during the system requirements analysis and architecture design activities. Standard uses a four-level software integrity scheme that is presented on Table 1.

Table 1. Different software integrity levels.

Criticality	**Description**	**Level**
High	The selected function affects critical performance of the system.	4
Major	The selected function affects important system performance.	3
Moderate	The selected function affects system performance, but workaround strategies can be implemented to compensate for a loss of performance.	2
Low	The selected function has a noticeable effect on system performance but only creates inconvenience to the user if the function does not perform in accordance with requirements.	1

Standard suggests that the minimum V&V tasks should be identified for the different integrity level schemes. The mapping of the software integrity level scheme and the associated minimum V&V tasks should then be documented in the software verification and validation plan. [4]

There are also other approaches for V&V planning that are used in the industry. For example, instead of looking into software integrity levels, one approach concentrates on a careful risk analysis of development work. In this approach, a risk analysis is performed on the content before starting each new increment development. The identified risks of the feature, interface or non-functional characteristics are evaluated and presented on a table (see the sample risks table, Table 2). The impact and the likelihood of occurrence are estimated for each risk, and this information is marked on the table.

Table 2. A sample risks table.

Risk description	**Probability**	**Impact**
Performance requirements	3	3
Failure in decoding data	4	2
System memory corruption	1	4

Based on the risk analysis, the test strategy for the increment is determined. For this purpose, a strategy matrix is created. The strategy matrix shows which techniques have to be used when testing different parts of the new increment. The required depth of the testing is also determined based on the strategy matrix. The Figure 6 presents a sample strategy matrix (adapted from [32]).

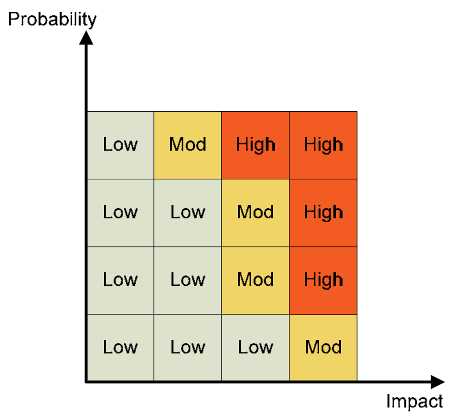


Figure 6. A sample strategy matrix.

The features, interfaces or non-functional characteristics are placed in the matrix based on their impact and probability ratings. The matrix then shows to which depth the testing should be performed. Those which are placed on “High” blocks have to be tested thoroughly; those in “Mod” blocks require only regular depth and those inside “Low” blocks need very little testing. For example, if a new feature has an identified risk whose impact factor is three and probability is two, this would be placed in the “Mod” block and would be tested with regular depth.

2.6.3 Approaches to testing

Earlier in this section, testing and V&V planning was discussed at the management level. At the lower level, when the actual cases for tests are planned, a deeper knowledge of the software internal functioning is required. In this sub-section, some typical approaches for designing and conducting test cases are covered.

Boundary value analysis and equivalence partitioning. The equivalence partitioning technique is based on identifying the value ranges of variables for which the component behaviour is equivalent. Thus, testing with one value in range suffices to get the information that would otherwise be obtained by testing all values within the range. Boundary value analysis focuses on the boundaries of the partitions identified during the equivalence partitioning process. These techniques can be applied for both input- and output variables and are test case design strategies in black box testing. [33, 34]

State transition testing. In state transition testing, the focus is on making the correct transitions from one state to another. This may require a lot of effort to check the component’s reaction to all possible events received in all states (both valid and non-valid). To minimise this effort, automated test generation tools have been developed for generating transition scripts. This is the black-box testing technique. [33, 34]

Cause-effect graphing analysis testing. In this technique, an analysis of the component’s specification is made. Based on this analysis, the component’s behaviour is modelled by means of causes and effects. Conditions and results are determined from the model and a decision table is established. This table will result in test cases. Cause-effect graphing is a black-box test design method. [33, 34]

Branch analysis and decision testing. Branch testing is based on testing decision points in the source code. The decision points are statements in the software, which may transfer execution to another statement. Typical statements are ‘if-else’-statements and for/while loops. In the ideal case, all branches of each statement in the source code are hit at least once during testing. This type of testing can be considered white-box testing. [33, 34]

3. Early V&V challenges

A major problem currently facing the software industry is the lack of tools to automatically validate and verify software in comparison to accepted requirements and environmental and operational constraints. There is also a lack of tracking tools to manage requirements or design changes that affect the approved characteristics of the product for internal company compliance rules. [35]

Another challenge for software organisations is that software testing and fixing bugs in the late phase of system development process is currently too time consuming and often leads to budget and schedule overruns. At least 50% of all software costs are spent on testing activities.[2, 36] Humphrey points out in [2] that it takes more than 40 times longer to find and fix a defect found in system testing than one detected in module testing. This is why it is important for V&V to be done early in the product development and as extensively as possible.

The more complex the systems become, the more significant the problems of testing will be. The time used in testing is one of the bottlenecks of the productivity of the software industry and causes most of the delays in time to market. Thus, the overall time used in the testing and in defect fixing should be decreased. [35]

This chapter addresses the current state-of-the art of early verification and validation. During this study, new techniques, methods and processes for improving early V&V process were found, and they are listed in this chapter. This chapter also presents the results on the survey on state-of-the-practice, which is an industrial survey conducted among the Evolve project partners.

3.1 Motivation for doing early V&V

Barry Boehm and Victor R. Basili wrote an article [3] that lists the top 10 software defect reduction methods for the product development process. According to the article, the authors discovered that the most important method for defect reduction is the ability to filter software problems during the requirements and design phase. The article states that "finding and fixing a software problem after delivery is often 100 times more expensive than fixing it during the requirements and design phase". The ratio can be more like 5:1 in smaller, non-critical software systems. [3] Still, this insight encourages companies to invest and focus on ways to improve the verification and validation process of early development phases. As Boehm points out in [1], "this insight has been a major driver in focusing industrial software practice on thorough requirements analysis and design, on early verification and validation, and on up-front prototyping and simulation to avoid costly downstream fixes". Boehm and Basili also noted that "about 80 percent of avoidable rework comes from 20 percent of the defects" and "about 80 percent of the defects come from 20 percent of the modules and about half the modules are defect-free”. As a consequence of this varied distribution of defects, it can be concluded that a focus on early defect detection pays off.

The concept of early validation is not new. In 1989, Wallace & Fujii [37] already recommended that “properly applied throughout the life cycle, verification and validation can result in higher quality, more reliable programs” and also that “ideally, V&V parallels software development”. They list tasks and methods to be applied in all lifecycle phases, from concept and requirements definition up to test, installation and checkout. However, for a variety of reasons, such as the problems in effectively evaluating incomplete components, defect identification techniques are often applied only towards the end of the development cycle.

Testing effort distribution in the late phases of the project follows an s-curve: it starts at the low level, ramps up in the test phase, and stays constant until the delivery date (Figure 7). In reaction, the number of identified defects increases locally, creating a “hump” profile. [38]

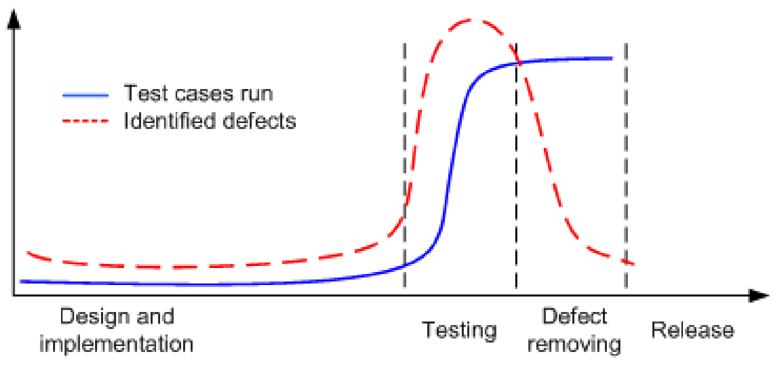


Figure 7. The S-curve and the “hump.”

Validation late in the project can be risky. It increases the possibility of missing deadlines or quality targets towards the end of projects, when there are also fewer opportunities to correct problems. Also, it reduces the process predictability. There are many valid reasons for late V&V, including [38]

- lack of effective methods and technologies (a completed system is easier to evaluate than its early artefacts (requirements, plans, models))
- incremental development issues (sometimes V&V cannot be applied before the system is complete)
- system architecture concerns – e.g. software/hardware co-development (V&V cannot be applied before the software is running on hardware)
- costs control – investments are needed to implement V&V activities during development; sometimes these investments are too large.

The importance and benefits of early V&V are generally understood, and methods and tools to achieve them have been proposed and implemented. The next section presents the results of the study on the most significant approaches described in recent scientific and technical literature.

3.2 Literature study on early V&V

As part of this master's thesis, two large studies were done to support the framework design. Firstly, the state-of-the-art of the early V&V was studied [38], and its results are presented in this section. Secondly, a survey on state-of-the-practice was carried out, which results are briefly analysed in the next sub-section.

The state-of-the-art (SotA) study was extensive. A total of six researchers contributed to the study; the author of this thesis was the main author. The SotA report consisted of over 50 pages, with references to over 80 different sources.

The SotA study was conducted in a structural way. First it defined the product lifecycle model and its phases. Then each phase of the life cycle was studied separately and the processes, methods and techniques for improving early V&V during these phases were identified. Then there was a study of the tools which are commonly used when doing V&V in general. Some features of the tools were listed and the tools were sorted based on their potential usage. The following sub-sections briefly present the study results.

3.2.1 V&V lifecycle model

In this section, the concept of software development lifecycle as well as the generic lifecycle model used for categorising V&V methods is presented. This lifecycle model serves as the basis for building the framework. The contents of this section are adapted from the SotA study [38].

The product lifecycle begins with a definition that describes the expected behaviour of the system. The solution is then designed, built and tested to compare behaviour with its initial description. If it passes, it then gets deployed; if it doesn't then it gets reworked until it does pass. Lifecycle artefacts are the by-product of this evolution. These artefacts represent the system at particular stages of its life. Requirements represent the behaviour, designs represent the solution, source code represents the implementation, and tests represent the qualifying argument for deployment. [39]

V&V is a process in itself and it has a lifecycle of its own. The V&V lifecycle is run in parallel with its development. For example, as the behaviour is defined and its by-product generated (e.g. requirements specification), V&V will perform the requirements analysis. Based on their assessment, the V&V process will gain an understanding of the system's behaviour. [39]

According to van Moll et al. [40], the development of a complex product can be considered an ordered sequence of phases and activities, regardless of the specific development approach (e.g. incremental, waterfall or evolutionary). In theory, each phase and activity is open to the injection of defects in the resulting information or work product due to many causes (Figure 8). [40] The injected defects add to those that have been already injected in the previous phases and could propagate in the subsequent phases.

To be able to get the system "right the first time", the number of injected defects should be minimised, which is the task of defect prevention measures. In reality, defects will occur. Therefore, for any given phase, defect detection measures should be taken. These measures must reflect the typical type of defects injected and the information or work product produced. The goal is to minimise the amount of defects that propagates in the subsequent phases. Ultimately, the number of residual defects in the end product should be as low as possible. [40]

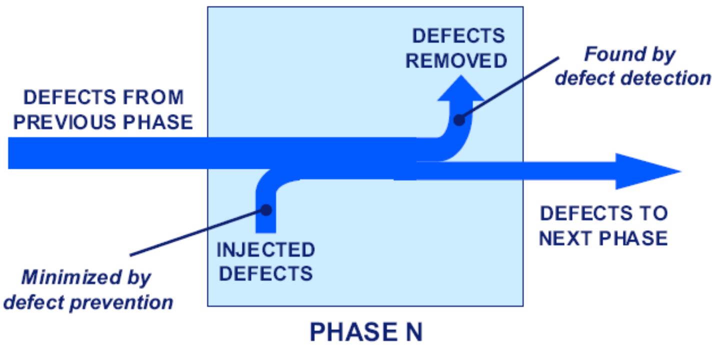
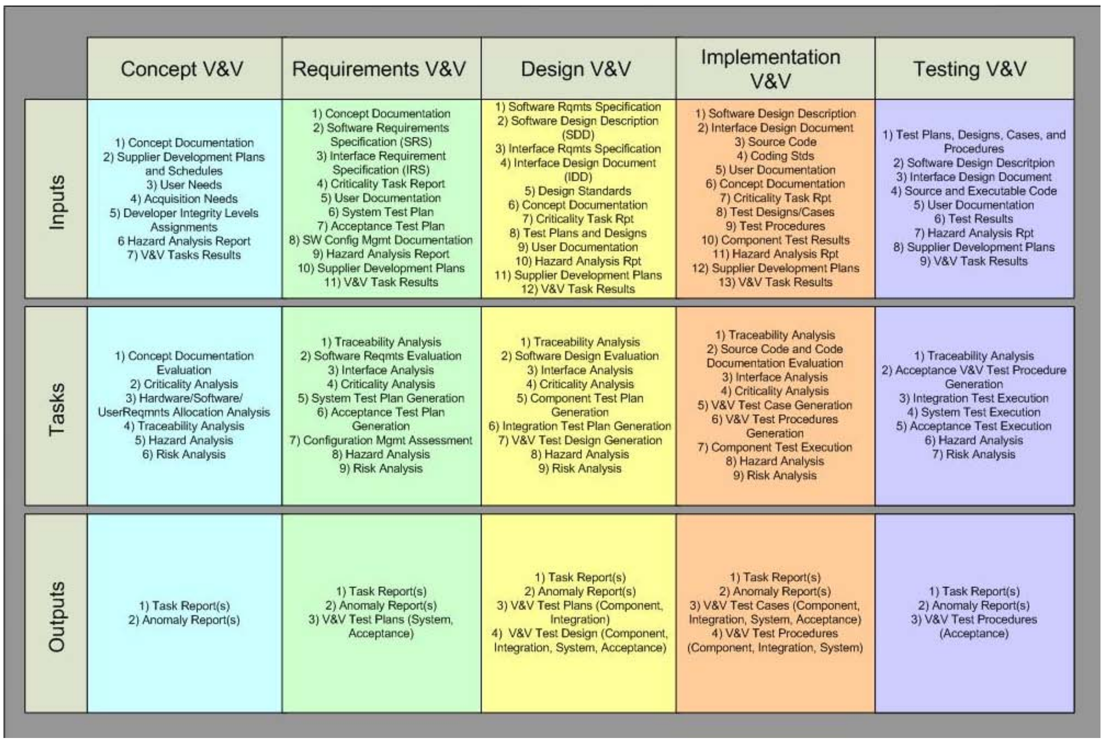

DEFECTS FROM PREVIOUS PHASE
DEFECTS REMOVED
Found by defect detection
DEFECTS TO NEXT PHASE
INJECTED DEFECTS
Minimized by defect prevention
PHASE N

Figure 8. Defect injection and detection in a phase.

The following software development life cycle description is adapted from the IEEE Standard for Software Verification and Validation [1, 4] (Figure 9). The graphic illustration provides a sample overview of the V&V inputs, outputs and tasks (at the highest software criticality level). Figure 9 represents the phases that generally take place during the lifecycle of a typical software development. The list of tasks within the phases is not comprehensive nor does it contain any mandatory tasks, but the tasks are included to help the reader get an idea of the activities that could take place during each particular phase.



	Concept V&V	Requirements V&V	Design V&V	Implementation V&V	Testing V&V
Inputs	1) Concept Documentation 2) Supplier Development Plans and Schedules 3) User Needs 4) Acquisition Needs 5) Developer Integrity Levels Assignments 6 Hazard Analysis Report 7) V&V Tasks Results	1) Concept Documentation 2) Software Requirements Specification (SRS) 3) Interface Requirement Specification (IRS) 4) Criticality Task Report 5) User Documentation 6) System Test Plan 7) Acceptance Test Plan 8) SW Config Mgmt Documentation 9) Hazard Analysis Report 10) Supplier Development Plans 11) V&V Task Results	1) Software Rqmts Specification 2) Software Design Description (SDD) 3) Interface Rqmts Specification 4) Interface Design Document (IDD) 5) Design Standards 6) Concept Documentation 7) Criticality Task Rpt 8) Test Plans and Designs 9) User Documentation 10) Hazard Analysis Rpt 11) Supplier Development Plans 12) V&V Task Results	1) Software Design Description 2) Interface Design Document 3) Source Code 4) Coding Stds 5) User Documentation 6) Concept Documentation 7) Criticality Task Rpt 8) Test Designs/Cases 9) Test Procedures 10) Component Test Results 11) Hazard Analysis Rpt 12) Supplier Development Plans 13) V&V Task Results	1) Test Plans, Designs, Cases, and Procedures 2) Software Design Descritpion 3) Interface Design Document 4) Source and Executable Code 5) User Documentation 6) Test Results 7) Hazard Analysis Rpt 8) Supplier Development Plans 9) V&V Task Results
Tasks	1) Concept Documentation Evaluation 2) Criticality Analysis 3) Hardware/Software/ UserReqmnts Allocation Analysis 4) Traceability Analysis 5) Hazard Analysis 6) Risk Analysis	1) Traceability Analysis 2) Software Reqmts Evaluation 3) Interface Analysis 4) Criticality Analysis 5) System Test Plan Generation 6) Acceptance Test Plan Generation 7) Configuration Mgmt Assessment 8) Hazard Analysis 9) Risk Analysis	1) Traceability Analysis 2) Software Design Evaluation 3) Interface Analysis 4) Criticality Analysis 5) Component Test Plan Generation 6) Integration Test Plan Generation 7) V&V Test Design Generation 8) Hazard Analysis 9) Risk Analysis	1) Traceability Analysis 2) Source Code and Code Documentation Evaluation 3) Interface Analysis 4) Criticality Analysis 5) V&V Test Case Generation 6) V&V Test Procedures Generation 7) Component Test Execution 8) Hazard Analysis 9) Risk Analysis	1) Traceability Analysis 2) Acceptance V&V Test Procedure Generation 3) Integration Test Execution 4) System Test Execution 5) Acceptance Test Execution 6) Hazard Analysis 7) Risk Analysis
Outputs	1) Task Report(s) 2) Anomaly Report(s)	1) Task Report(s) 2) Anomaly Report(s) 3) V&V Test Plans (System, Acceptance)	1) Task Report(s) 2) Anomaly Report(s) 3) V&V Test Plans (Component, Integration) 4) V&V Test Design (Component, Integration, System, Acceptance)	1) Task Report(s) 2) Anomaly Report(s) 3) V&V Test Cases (Component, Integration, System, Acceptance) 4) V&V Test Procedures (Component, Integration, System)	1) Task Report(s) 2) Anomaly Report(s) 3) V&V Test Procedures (Acceptance)

Figure 9. V&V tasks and the different phases of the product lifecycle.

In different companies or development environments, these phases may have different names; they may contain different tasks or they may be applied in a different order. However, it can be said that these phases form the backbone of the development lifecycle of practically all industrially produced software-intensive systems.

3.2.2 Techniques, methods and processes

The previous section presented five phases of a V&V lifecycle. Four of these phases were used in the state-of-the art study to introduce currently available methods, techniques and processes intended for early V&V. The concept V&V and the requirement V&V phases were combined in the study, and this combination will be called the requirements definition phase. In this sub-section, the techniques and methods identified in the study are listed. Each of these items is presented separately in the SotA document [38]. The items were selected so that the basic V&V activities (see the previous chapter) would also be covered as thoroughly as possible.

In the study, the methods and techniques were categorised based on the lifecycle phase so that the best ones (some of which could be applied also during the other phases of the development) could be applied. The methods and techniques are listed in Table 3 and are organised in the alphabetical order under corresponding phase.

Table 3. Early V&V methods and techniques.

Requirements definition phase
B-method
Booch Methodology
Jacobson Method
Model-based Requirements Specification and Validation
Petri Nets
Requirements Generation Model (RGM)
Scenario-based Requirements Engineering (SCRAM)
Sequence of thinkLets
Specification Language Z
Storyboard Prototyping
Rapid Prototypes
Vienna Development Model (VDM)
Volere Method
Design phase
Abstraction and model transformation
Design-for-testability
Inspections
Model-based Verification
Model checking
Model reviews and modelling guidelines
Technical reviews
Walk-throughs
Implementation phase
Code generation
Code inspection
Code smells
Continuous Integration
Defect detection in editor
Static Analysis
Unit Testing / Test Driven Development
Verification of software via Integration of Design and Implementation
Testing phase
Model based integration and testing method
Model-based testing
Mock Objects

Amongst these techniques, some will be selected for this thesis. These techniques are presented later in chapter 5.

3.3 Survey on state-of-the-practice

One objective of this thesis was to design and conduct an industrial survey among the Evolve project partners. The survey was drawn up in order to get information about state-of-the-practice in the industry, i.e., the verification and validation tools, processes and methods the industrial partners of the project are using. Another goal was to collect information regarding the challenges that the partners are facing during their verification and validation (V&V) processes. The challenges that came up in the survey are addressed and novel solutions are developed during the Evolve project. The results of this survey are thoroughly described in document [41], which was written by the author of this thesis. That document is published as a white paper that was designed to directly guide the research work done for this thesis and later on in the Evolve project. The results that are most relevant to this work are presented here.

3.3.1 Survey background

The survey was intended for all industrial partners of the Evolve project. The partners were requested to select participants for the survey within their company; specifically, it was intended for the hands-on practitioners of software development. Since the questionnaire contained several questions on the general arrangements of development work — such as tools, languages, and methods — it was not considered practical to ask everyone in the large project to participate in the survey. Instead, it was suggested that the contact person in the company could select a few key persons per project to answer the survey, thus providing a cross-section of typical project settings.

The questionnaire used in the survey was divided into seven parts. The first four parts were traditional V&V process-related questions and were destined for all respondents. The latter three parts were reserved for respondents involved in model-based development. The survey was formed as follows: general information (section 1), development environment and practices (sections 2–3), V&V activities (section 4) and model-based development activities, performance and usability (sections 5–7). The questionnaire contained both structural (multiple choice) and non-structural (open-ended) questions. In order to be sure that the structured questions could be answered in all cases, respondents could select “Other” as an answer and type their responses in an open text field. Also, to ensure that the respondents could include everything they wanted to say about the issue, there was an open “Comment” field at the end of the each section where respondents could add additional information.

The questions for the survey were worded carefully to avoid misunderstandings. In preparation for the survey, the following reports were used:

- test processes in software product evolution – a qualitative survey on the state of practice [42]
- a State-of-practice questionnaire on verification and validation for concurrent programs [43]
- verification and validation in industry – a qualitative survey on the state of practice. [44]

3.3.2 Survey implementation

The data for the study was collected using a web-based questionnaire. The body for the questionnaire was made with the Adobe LiveCycle Designer tool, so the file format was Adobe PDF. Using a PDF form has both advantages and disadvantages. Using PDF for surveys is quite laborious and some programming and scripting skills are needed to make the survey work reliably online. Once completed, however, the result can be a professional-looking layout that is easy to use, and the working base can be used in future queries. The use of a PDF file also offers the possibility to print out paper copies that can be compared side by side with the online query to get responses to the survey offline, for example in workshops or seminars. In addition, the built-in security properties of the PDF format were considered useful for this purpose. For example, automatic “spamming” applications are currently incapable of filling in PDF forms.

The questionnaire form was placed on a web server. The form offered respondents the chance to submit their answers over the network. To make the online submission work, a Perl CGI script had to be implemented. The script collects the respondents’ answers one by one and saves them as XML files on the server; it also makes it possible to save the answers in a MySQL database. It is easy to sort the data from a MySQL database with some basic SQL queries. Furthermore, it is possible to import the data to MS Excel, which enables tables and other visual presentations (such as customised graphs) to be created.

The questionnaire was available to respondents for two months, between July 6th–September 9th 2009. During that time, a total of 16 responses from 11 different companies were received. As the goal of the survey was to collect partners’ challenges and practices, and only the key persons from the projects were selected to answer, the amount of responses can be considered satisfactory for the purpose of this thesis.

3.3.3 Respondents’ background

In the first section, the respondent’s background information was collected along with his/her company information. This information is relevant for the analysis of the survey results.

The respondents were mainly experienced professionals as the majority had at least two years of experience with software development and none had less than three months of experience. According to the answers, the product domain of the companies in the survey was in almost all cases “Embedded Software: telecommunications, automotive, electronics”. Only one participant reported his company’s domain area to be “Software systems: Information systems, internet applications”.

3.3.4 Development practices and environments

Parts 2 and 3 of the survey included questions on software development practices and the environments that the partner companies are using. This information was relevant when analysing the results of the survey. It was also used when determining which tools would be studied further during the work of this thesis.

The respondents were asked about how the software development process is defined at their organisation. Most responded that their organisations have guidelines and templates for almost all phases of SW development.

In addition to the software development process definition, the participants were asked to characterise what kind of product lifecycle models are applied in their organisation. This question was formulated as multiple choice — indicating the use of more than one lifecycle model. The most popular lifecycle models among the participating companies were incremental/iterative and agile processes. Although both of these models were selected by nine respondents, they are not connected in all cases. Traditional process models are also applied in many participating companies; the V-model is used by six and waterfall model by three respondents.

The development environments of partner companies were studied in the survey by asking questions about the languages and the tools/environments that they use during different phases of product development. The product development process was divided into five generic phases and separate questions delved into the details concerning each phase. For each answer, there was an open text field on the form. The process was divided into the following phases: requirements definition, architectural design, detailed design, implementation and testing. The overview of the tools used by the participating companies in different phases can be found on Table 4.

In the requirements definition phase, all participating companies reported using natural language for presenting requirements. Five of the respondents stated that UML is used for use case definition. The tools that are used in this phase are mostly Microsoft Office tools (mainly MS Word and MS Excel), but Matlab, Simulink and Enterprise Architect are used by some respondents.

The design phase was divided into architectural and detailed design phases, but it seems that with regard to the technical environment, participating companies do not clearly distinguish between these two phases. Thirteen respondents reported that they are using UML, SysML or some object-oriented and structured languages (e.g. Matlab) during these design phases while others (three respondents) have no specific languages in use. In these cases, the design is described in natural language. The tools that are most popular among the participating companies are Enterprise Architect and Matlab, both reported five times. Other tools mentioned are Microsoft Visio, Telelogic Rhapsody, and LabView. Four respondents stated that they have no specific tools in use for design phases.

The partner organisations seem to be quite flexible when it comes to the implementation language. Many participants reported using multiple languages in their organisation, and some even stated that they are capable of using any available programming language, if needed. According to the answers, C or C++ is used by the organisations of all but one respondent, while Java was reported by seven respondents. Other languages that were reported are Python, PHP, PERL, Matlab and Ada. The most popular tool for implementation is Eclipse, which eleven respondents named. Emacs was used by four respondents and Visual Studio in two cases. Some respondents stated that they also use “a good text editor” during this phase.

In the implementation, C and C++ are the most used languages for testing as well. XML, TTCN-3, Python and Java are other languages that are used in two or more cases. Many participants reported that they are using self-made or company-specific testing tools. There seems to be no single testing tool that takes preference above the others. Different kinds of test benches are used in four cases and LabView and Eclipse-based testing in two cases.

The following Table 4 summarises the results of this section. This table shows the number of cases in which a respondent mentioned a tool used in his/her organisation during each product development phase. For each phase, the tools used the most were highlighted in green, except for the testing phase, where the number of tools used was significantly low. Some of the respondents pointed out that since they are not part of the development team, they have indicated the tools that they have seen the development teams using. It was also stated that there is such a variety of tools available that it is almost impossible – and obviously unnecessary – to list them all.

Table 4. Tools and their usage in the different product development phases.

Tool \ Phase	Requirem.	Arch. design	Det. design	Implement.	Testing
Microsoft office	13	2	1		
Matlab / Simulink	1	4	5	1	1
Enterprise Architect	2	5	3		
Eclipse			1	11	2
Emacs			1	4	
Visual studio			2	2	
Rhapsody		2	2		
DOORS	3				
LabView				1	2
CxxTest					1
Cantata++					1
C++Test					1
Jira	1				
Latex	1				
IRqA	1				
VI				2	
Melexis Tools				1	
Borland C++				1	

3.3.5 Validation and verification activities

In the fourth part of the survey, the participants were asked about verification and validation activities and how they are commonly applied in their projects. The respondents' views on the effectiveness of companies' V&V process were also given along with the greatest challenges in the V&V process.

Figure 10 shows the amount of different V&V activities and techniques reported by the respondents at the participating organisations. The activities have been grouped as follows: green bars indicate the activities related to testing, which was the biggest group. Pink bars are simulations and prototyping relating activities; brown bars show the review related activities. Blue bars indicate the use of code/model analysis tools, while the last two bars (in purple) show the use of pair programming and test-driven development.

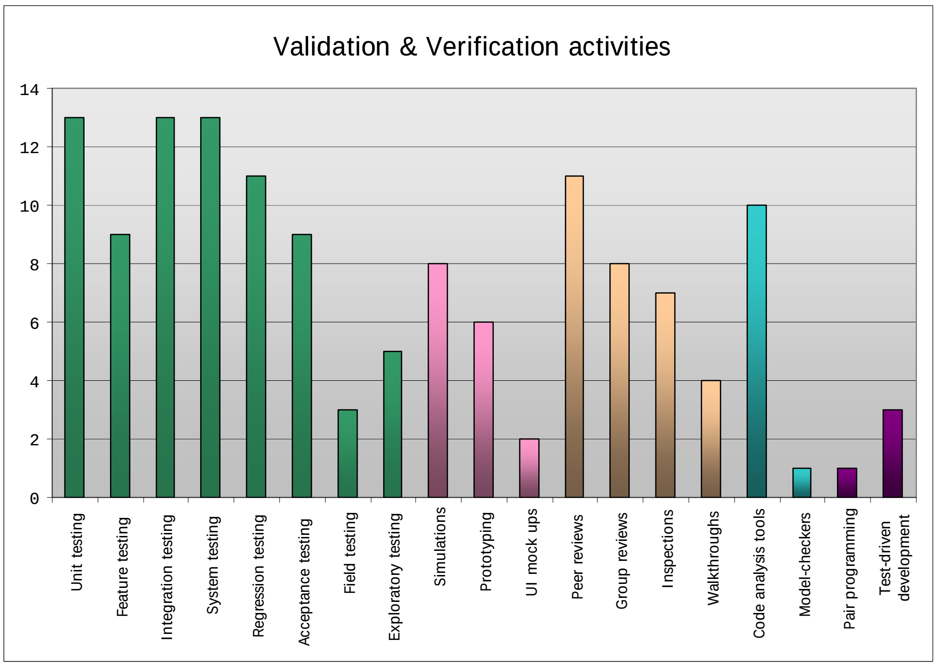


Figure 10. V&V activities and their usage in the projects.

As can be seen on Figure 10, testing appears to be the most popular technique used in verification and validation. Unit testing, integration testing and system level testing are the most applied testing activities amongst the partner organisations – each selected by 13 respondents. Eleven participants reported the use of regression testing, while acceptance testing and feature testing were both reported by nine respondents. Reviews also seem to be very common activity according to the survey participants. The most common review technique is (technical) peer review with 11 responses. Group reviews and inspections are done by about half of the respondents, with eight and seven responses respectively. Code analysis tools are used quite often, with a total of ten responses.

Guidelines and templates for V&V activities seem to be available in participating organisations in comparison to guidelines for software development practices. Eleven participants reported that they have some guidelines and templates for V&V while five indicated that they have fully documented guidelines and processes. All of the respondents reported that they have some guidelines and templates in use. Nine of the participants reported that they have also had training on verification and validation (e.g. testing), while seven reported no such training.

With regard to the scope of testing, the participants' responses varied significantly. Five participants reported that testing is always performed with a similar scope and all parts are tested. Four answered that all parts are tested but newly added features are emphasised. Another four respondents reported that only new features are tested, and regression tests are run. There was also possibility to select "other" as the answer and describe the activities in their own words; three participants selected this option. Two of them answered that new features are tested, but no standard regression tests are done. One person pointed out that his company uses all of the options listed in the question, depending on the project.

The participants were also asked to assess how effective they considered software testing in their company. Figure 11 shows how this question was answered by each represented role. The majority of the respondents considered their software testing to be effective; according to eight participants, software testing is quite effective in their company and one even answered that it is very effective. Four persons — three of them developers — felt that software testing is quite ineffective and three participants had no opinion on this question.

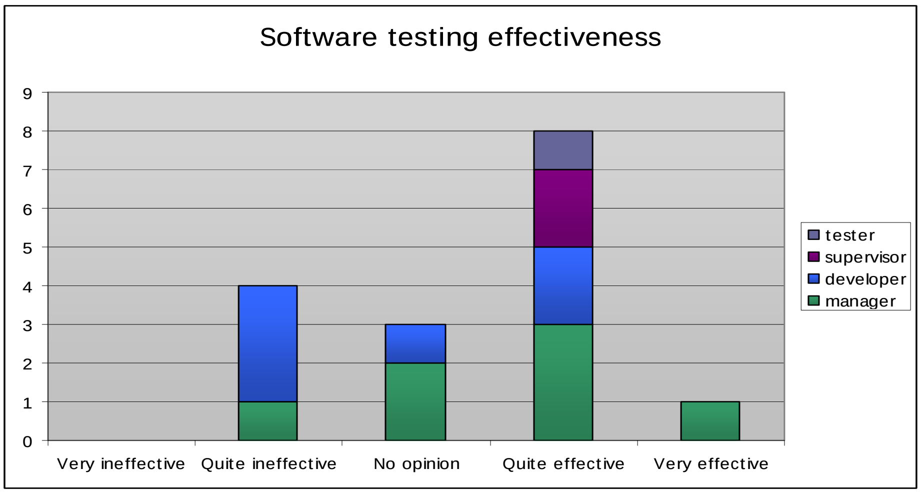


Figure 11. Software testing effectiveness and roles in the company.

Participants were requested to list the greatest challenges in V&V of their companies. There were three recurring challenges in the answers. The first challenge is **lack of resources**; there are not enough resources/time scheduled for the V&V tasks. The second – which can be seen as related to the first one – is the **lack of test automation**; according to the respondents, too much effort is spent on repeating the same tests. The third challenge that can be considered recurring is the **traceability of requirements**; respondents commented, for example, that the requirements and the design often do not accurately reflect the item being tested, so time is wasted creating proper test specifications. Other challenges that were listed are reaching full code coverage in tests and insufficient knowledge of testing (e.g. when developers are running the tests themselves).
As pointed out by Boehm [1] in 1987, the later the SW defects are found, the more expensive they are to fix. In order to gain insight on potential methods and techniques to be applied in the later stages of the Evolve project, the respondents were asked what could be done to acquire information on the quality of the product earlier in the development lifecycle. The respondents suggested the following:

– more automated testing in earlier phases
– early models that can incorporate suggestions and tinkered with
– fully defining V&V procedures from the beginning of the project
– thinking about testability during the requirements definition
– validating regularly from the very beginning of the product development (e.g. scrum development style)
– creating tests before writing the implementation code

- involving SW people from the very beginning, from the definition of the hardware
- applying the agile development methods instead of the traditional method
- recording the metrics from all projects and using the historical data to plan and predict the possible amount of deviations and defects at the early stages
- making test results visible to those that need to make the decisions.

3.3.6 Model-based testing

In the next section of the survey, respondents were asked about the model-based verification and validation activities in their companies. One respondent stated his company does not use model-based testing, so this section had only seven responses. Model-based testing is used quite evenly in the different testing phases by the participants' companies (Figure 12). Integration, system and regression testing were each selected three times and unit testing, twice. In most of the cases the tests are executed manually. Two respondents reported that the model-based testing is automated to some degree in their company.

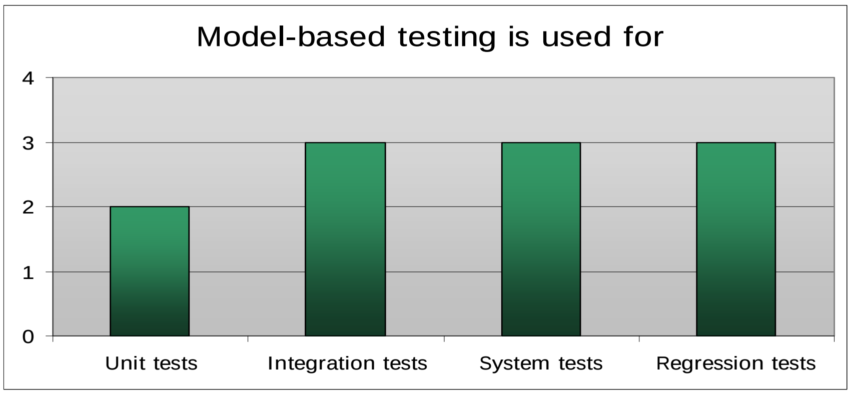


Figure 12. Use of model-based testing in different testing phases.

3.3.7 Summary of results

The purpose of the survey was to gain insight into the state-of-the-practice of the project's industrial partners. This information is needed when planning the framework, in order for it to better support the industrial partners' needs. The survey covered various topics related to software development — the current state-of-the-practice was studied with regard to the systematic nature of development practices, tools and environments, verification and validation practices, as well as model-based software development. Furthermore, the respondents were asked about their views on current challenges as well as the potential opportunities in early V&V.

In general, respondents felt that the software testing practices applied by their company are effective. However, they could also see room for improvement, and believed that there are concrete steps that can be taken in order to further develop the current procedures. The challenges faced in the V&V process of the partner companies included a lack of resources, a lack of testing automation and insufficient traceability of the product requirements. To improve the V&V process, the partners mentioned ways to help manage product quality during the early phases of the development lifecycle. These issues are potential research subjects in the Evolve project and will be considered when defining requirements for the framework later in this thesis.

Even though the number of responses in this survey was not excessive, the answers were provided by a competent body of respondents. The respondents clearly form a heterogeneous group with regard to their country of origin, their role in their respective organisations and in the project, as well as the size of the company they represent. This means that the responses can be considered representative of the views of the industrial partners participating in the Evolve project. Therefore, this survey provides valuable data concerning the current state-of-the-practice of V&V in software development in the European setting. In this work, the results of the survey will be used in defining the framework in the following chapters.

4. Requirements for the framework

One objective of this thesis work was to collect requirements for the Evolve framework. Since the requirements were gathered mainly through the survey and from the literature study, the actual work for the requirement collection was presented in the previous chapter. In this chapter, requirements are listed and described for the Evolve framework as a whole. Since this framework is planned to be implemented only partially in this thesis, solely the requirements pertinent to this thesis needed to be selected. At the end of this chapter, the requirements for this work are identified.

4.1 Requirements collection

Three sources were used to collect the requirements for the framework. In this section, we present how these sources were used.

The survey on state-of-the-practice [41] and its results were presented in the previous chapter. Different sections of the survey were analysed and by summarising the results, some of the most important issues were suggested as the requirements for the framework. The respondents were also asked about the main challenges in the V&V process of their companies and what could be done to obtain information earlier in the development lifecycle. Many requirements were formed based on these findings.

Plenary workshops are the Evolve project's internal meetings where the project partners gather to plan the issues at the consortium level to guide the work done in the different work packages. In one of these workshops, the participating members were asked to discuss, list and present some of the issues they would like to see included into the Evolve framework. These issues were collected and the resulting list was analysed. Based on this analysis, some of the issues were selected for the framework's requirements.

The state-of-the-Art (SotA) [38] literature study on early V&V processes, methods and techniques was also presented in the previous chapter. The SotA was used in requirements definition as a supportive source. It was a kind of confirmation for the selection of some requirements that were found in one of the two other sources. This confirmation was done by determining if the issue in question was considered important in the literature.

4.2 List of collected requirements

Table 5 lists the collected requirements for the Evolve framework. Requirements have an identification (ID) number which can be used when requirements are mentioned. The source of the requirements is one of the abovementioned sources. On the table, the sources are referred to with the following abbreviations:

S = Industrial Survey on State-of-the-Practice
W = Project Plenary Workshop
A = State-of-the-Art study on Early V&V.

The table shows the requirement names and short descriptions for the Evolve framework. The requirements that were selected for this thesis are indicated in purple. The next section further discusses the requirements for this work and the basis for their selection.

During the later stages of the Evolve project, the requirements presented here may be emphasised and some new requirements may be defined. Some of the presented requirements may partially overlap. For example, R013 is a special instance of R003, but it was decided that both of these should be present to emphasise the agile aspects of incremental development while still maintaining the other aspects.

Table 5. Collected requirements for the Evolve framework.

Requirement ID	Source	Requirement name	Requirement description
R001	W	Work guidance	The framework shall help developers keep work on track during the development to guide them in applying early V&V activities.
R002	S, W, A	Product test generation	The framework shall help in the transition from model verification tests to actual product tests.
R003	S, W	Solutions for iterative V&V	The framework shall be a technical tool that helps in iterative V&V.
R004	W	Model consistency	The framework shall allow developers to keep their architectural models up-to-date at all times.
R005	S, W, A	V&V reusability	The framework shall help developers reuse models and code with V&V scenarios from other projects.
R006	W	Adaptability	The framework shall have the flexibility to provide different implementations for different companies/situations.
R007	S, W	Progress indication	The framework shall indicate project progress based on development artefacts and metrics.
R008	S, A	Test automation	The framework shall enable more automated tests in the early phases of the product development.
R009	S	Process definition	The framework shall help fully define V&V procedures from the beginning of the project.
R010	S	Traceability of requirements	The framework shall improve the traceability between requirements and V&V activities.
R011	S	Early error detection	The framework shall help find software defects early in the project.
R012	S	Test result reporting	The framework shall make test results visible and readable to those that need to make decisions.
R013	S	Support for agile development	The framework shall support agile development methods.
R014	S, A	Statistical analysis	The framework shall support recording the metrics from projects and using the historical data to support planning and predicting of the quantity of deviations and defects of similar projects during early stages.
R015	S, A	Test case generation	The framework shall provide support for creating tests before writing the implementation code.
R016	S	Regular validation	The framework shall regularly validate the product from the very beginning of the development.

4.3 Requirements for this work

After the requirements were gathered, an internal workshop was arranged to review the requirements and selecting those which should be tackled during this thesis. During the selection, it was decided that this work would be a starting point for further developments in the Evolve project.

Seven out of sixteen requirements were selected for this work and some others (like R003 and R013) are partially covered. Since this work should be starting point for future development of the project, the selected requirements are mostly very basic ones, which help to define the structure for the framework. The R001, R006, R009, R010 and R016 requirements can be considered within these categories. R011 was selected to keep in mind the main idea behind the framework, even though this requirement was obviously not fully covered in this work. Requirement R015 was a bit more detailed than the others, but it was selected to give more specific content to the framework at this point. In addition, since there were no requirements selected for testing, R015 was selected to cover that phase.

5. Structure for the framework

Based on the requirements for the work defined in the previous chapter, work began on the framework design. The requirements provided both definitions and restrictions for the building process. In this chapter, the structure for the framework is presented. First the framework is formed and the formation is presented. Some of the methods found in the state-of-the-art (SotA) [38] study have been selected for this thesis. The selected methods are presented and the reason for their selection is explained in the second section. After the explanation of the methods, some tools that were selected to be used in the framework during this thesis work are briefly introduced. In the end of the chapter the selections that are made on methods, techniques and tools are summarized and the combined framework, which will be used in this work, is presented.

5.1 Formation of the framework

This section presents a conceptual model of the Evolve framework in its current state. The requirements R001, R006, R009, R010 and R016 were used to guide the planning and building of the framework. In the following, some of the main issues that were considered during the process are presented. The influence of the abovementioned requirements can be seen in these considerations.

The conceptual level model of the framework is presented in Figure 13. As shown in the figure, the framework is a toolbox-like combination of activities, methods and techniques for applying early V&V. The framework is planned so that it can be adapted for varying implementations at different companies and situations. Flexibility was considered important for supporting the development work at different software integrity (criticality) levels. There were multiple techniques and methods for early V&V that were found during the SotA study. As can be seen in the figure, these techniques and methods are used to form a backbone for the framework. The model of the framework was drawn up to help in selecting V&V methods and techniques to be applied in the different phases of the development.

At this point, the framework does not contain any predefined tools, but some of the methods and techniques (for example static analysis or model checking) require certain kind of tools to be used. In these cases the framework user can select the tool that suits best in his situation or in his environment.

In the framework model, the product development process is divided into four phases. As in the SotA study, these phases are requirements definition, architectural design, implementation, and testing and integration. Early V&V methods and techniques are listed for each of these phases similarly to the list presented previously in this thesis (see Table 3 in Chapter 3). The descriptions for these methods and techniques and the guidelines for using them are given in the SotA study. The common activities related to the development are also listed and shown in the model.

Of the techniques and methods presented in the figure, some are selected for testing as part of this thesis during the industrial case study. This case study is presented later in the Chapter 6. The selected techniques/methods are indicated in bold face in the model; as can be seen, they were selected from each development phases. This selection was made to be able to test the regular V&V of products, which starts from the beginning of the development cycle.

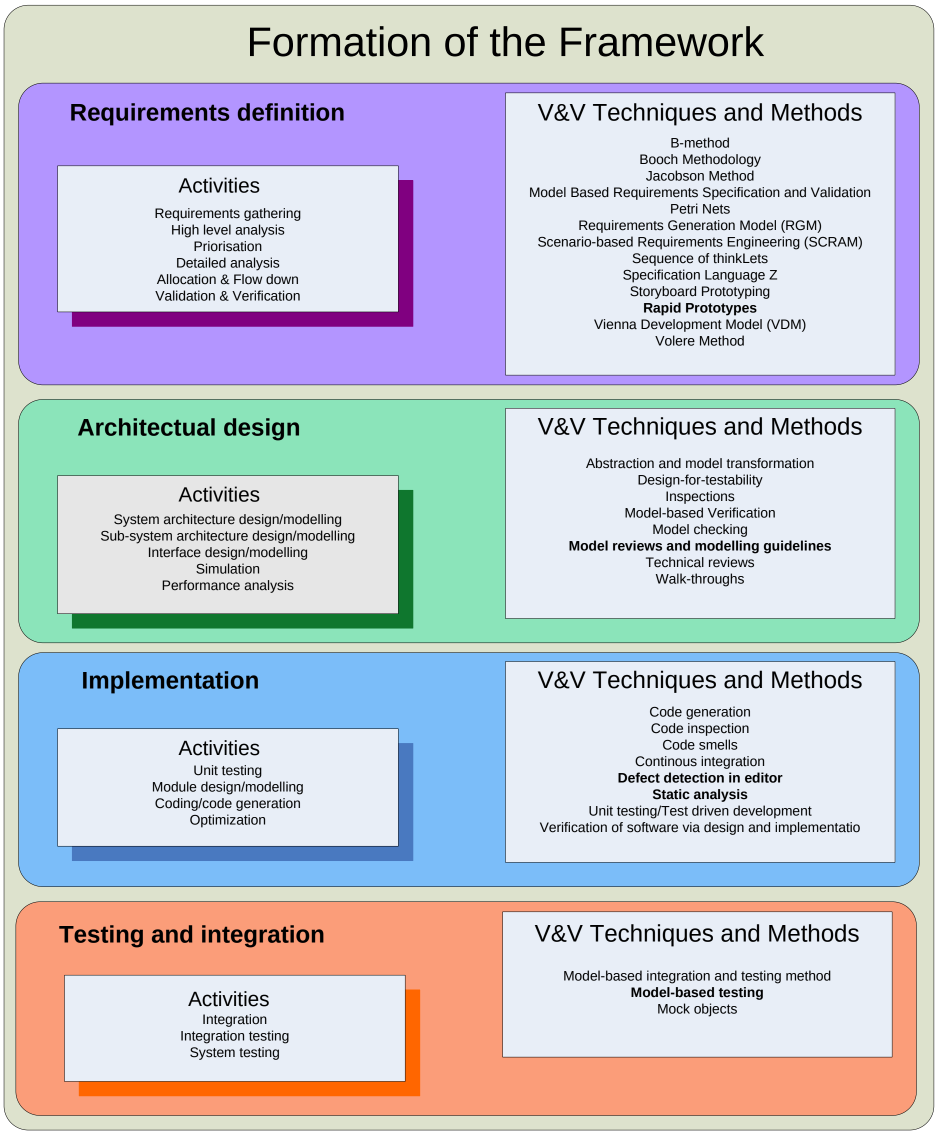


Figure 13. A conceptual level model of the Evolve framework.

5.2 Techniques and methods

As explained in the previous section, some of the techniques and methods from the framework are selected for this thesis. In this section, these are briefly presented and the following information about them is listed:

- phase(s) applied – one or more phases from the development cycle, where the technique can be applied
- V&V activity type – the basic V&V activities applied in the technique (see chapter 2)
- description – description of the technique (adapted from the SotA study [38])
- rationale – the reason for selecting this technique for this work.

5.2.1 Rapid prototypes

Phase(s) applied: requirements definition

V&V activity type: prototyping, reviews

Description: The use of rapid prototypes for early validation can accelerate product development in all phases of the development life cycle. The most obvious benefit is the ability to evaluate requirements for applicability and unanticipated errors early in the development life cycle. However, additional benefits can also be obtained during the follow-on phases of the development life cycle. [45]

A rapid prototype is a model implemented to evaluate a particular set of characteristics or attributes. The purpose of the rapid prototype is to ultimately serve as a standard to judge its source requirements. Rapid prototypes can be used early in the development of a system to identify and resolve validation issues. [45]

Rapid prototypes can be used to accelerate the discovery of a class of specification errors that were formerly “undiscoverable” in the early phases of development. Discovery of validation issues during the early phases of the development process drives the requirements to maturity sooner. Because an executable prototype is available and the requirements are mature, the development of verification test procedures can also be completed much sooner. [45]

Rationale: In the case study for this framework (will be presented in Chapter 6), the development is done for the client and only one meeting with client is arranged in the beginning. The requirements for the case are to be defined based on the customer’s verbal description of the case that is given in this single meeting session. Having rapid prototypes can help ensure that the customer’s requirements are understood correctly, before drafting a formal requirement plan and starting the actual technical design work. Many other options were also evaluated, but it was decided that for validating the requirements for this work, this single requirements validation technique sufficed. After the definition, the requirements will be reviewed in over-the-shoulder review.

5.2.2 Model reviews and modelling guidelines

Phase(s) applied: design phase

V&V activity type: reviews, simulation

Description: Model reviews and modelling guidelines are methods to ensure the accuracy of the model. Reviews can be informal or formal, and they can focus on architectural models and on design models. The intention of the reviews is to check models manually and ensure the correctness of the model.

Modelling guidelines are needed, especially when using general purpose modelling language that focuses on a wide variety of domains. When using a standard language that is not focused on narrow domain areas, it is not clear what different elements mean and which diagrams should be used in a particular domain. Modelling guidelines should be defined to help designers model in a uniform manner. That is why process activities like model reviews and checks are necessary to ensure that the model is described according to modelling guidelines. Modelling rules and design guidelines can help improve the quality of a model. Modelling rules can be expressed in either natural language or formal language, and they can also be implemented in the language, that is, in the metamodel of the language [46]. Design guidelines usually help to improve the quality of the system in a more indirect way. Following a set of design guidelines usually helps to improve the clearness and the readability of the code fragment or mode. It is also possible to check modelling rules and design guidelines automatically. Many modelling tools provide means for automatic model validation. Model validation, or consistency checking, is used for evaluating models with respect to the semantic and syntactic quality criteria. The model is compared to its metamodels and/or some defined constraints. The most commonly used constraint language is the Object Constraint Language (OCL), a formal language used to describe expressions on UML models. [47]

Rationale: Since the code generation is not used in this work, detailed design models are not needed. This means that techniques like model checking and model-based verification are not appropriate for the case. Model reviews and modelling guidelines was selected to help divide the problem into logical units. The implementation of the classes will be made according the models provided.

5.2.3 Defect detection in editor

Phase(s) applied: implementation

V&V activity type: unit/component testing

Description: The traditional approach to coding involves typing text in an editor, attempting to build, interpreting the error codes produced by the build tool chain, and fixing the errors, until an executable (build) is produced. Next, the build is run and tested, and the cycle is repeated until the desired behaviour is obtained. This process can be time consuming especially when the product is very large.

IDE tools provide text editors that check the code’s syntax in real time – while it is being typed – and highlight errors/warnings (e.g. by underlining). Defect detection in editor helps syntactically correct code to be written in a single step, and also reducing compiler “warnings” – as they become visible and easy to fix. It allows the developer to focus on runtime correctness rather than syntax.

Rationale: No code generation is used in this thesis, and the coding is done in a traditional way. As described above, in the traditional development, detecting defects in the editor is a good way to shorten development time and effort. This is also very easy to test, since for example Eclipse and Visual Studio development environments support defect detection by default.

5.2.4 Static code analysis

Phase(s) applied: implementation

V&V activity type: reviews, unit/component testing

Description: Static analysis refers to various techniques for the automatic identification of defects and faults without running the application (therefore, “static”). Static analysers work by parsing the source code and running formally defined rule sets against it.

Static analysis provides a fast and inexpensive alternative (or complement) to code inspection. While more prone to false positive and false negative findings, it can be run frequently on entire code bases. It detects faults less obvious to human readers like duplicate code. Louridas [48] introduces the main concepts and available tools for static analysis. His article concludes that “No machine can substitute for good sense, a solid knowledge of the fundamentals, clear thinking, and discipline, but bug detection tools can help developers.”

Rationale: Because there was only one developer doing the implementation, adequate code inspection would have been hard to arrange. Static code analysis is a fast and easy way to cover the most obvious faults in code. There are also some good static analyser tools available, with which our research team already had experience, so support was readily available if needed.

5.2.5 Model-based testing

Phase(s) applied: testing

V&V activity type: unit/integration/system level testing, black-box testing

Description: Based on the literature, a model-based testing term is used for a wide variety of test generation techniques. According to [49], model-based testing is the automatic generation of efficient test procedures/vectors using models of system requirements and specified functionality. Basically, the benefit of the model-based testing (MBT) is that it allows the test engineer to focus on specifying and modelling the behaviour in question with a high level of abstraction, instead of doing laborious manual test-case design, scripting and manual test execution.

In MBT, the test designer writes an abstract model of the system under test (SUT), and then the model-based testing tool generates a set of test cases from that model. Typically, a test tool is used to automatically execute these. The benefits that can be achieved in MBT are reduced test design time and the possibility to generate a variety of test suites from the same model simply by using different test selection criteria. [50] Thus, the effectiveness of model-based testing is primarily based on its potential for automation. The key advantage of MBT is that the test generation can systematically derive in all combination of tests associated with the requirements represented in the model to automate both the test design and test execution process [51]. Because the tests are generated from a model of the application, the model only needs to be updated in order to generate new tests when the application changes.

According to [50], building the tests using MBT can be divided into five main phases. These phases are

1. modelling the SUT and/or its environment
2. generating abstract tests from the model
3. concretising the abstract tests to make them executable
4. executing the tests on the SUT and assigning verdicts
5. analysing the test results.

MBT can be applied to any of the testing levels involved in V&V, such as unit testing, component testing, integration testing and system testing. Concerning the tests, functional testing, robustness testing, performance testing and usability testing, the main use of model-based testing is to generate functional tests. Since the test suites are derived from models and not from source code, model-based testing is usually seen as one form of black-box testing. The most common practice involves using black-box testing techniques to design functional and robustness tests. MBT is not widely used for performance testing, but this is an area under development [50].

Rationale: In the SotA study, few techniques were found that could be applied in the testing phase of the development. Of the found ones, MBT was the obvious choice for the framework. First of all, MBT covers requirement R015. In addition, the selection of MBT was supported by the fact that it is commonly used in the industry (also by the Evolve project partners). There are also some mature tools available for MBT, which have been studied in a previous study done by our research team members.

5.3 Tools

During this thesis work, the framework will be tried out in an industrial case (see Chapter 6), and some tools are needed during the case development work. As said before, the framework, does not define any tools that should be used with any method or technique and the tools should be selected case by case. In the industry the selection would be made based on the environment and existing tools that are used. Therefore, while selecting the tools for the framework in this work, the main criterion was their use by the project partners. This usage become clear from the results of the industrial survey (see Table 4 in chapter 3). In some cases, though, there were no specific tools mentioned that could be used in some specific area (for example, the static analyser tool). In these cases, the selection was based mostly on the assessment of the V&V tools in the state-of-the-art study [38] and issues like the suitability for the case and the tool's availability (e.g. open-source vs. commercial tool). In this chapter, the tools that are selected for the framework are presented and the reason for their selection is given.

5.3.1 Enterprise Architect

For the design phase, two tools were considered: Matlab/Simulink and Enterprise Architect (EA). According to the survey presented in section 3.2, both of these tools were equally used by the project's industrial partners. Matlab and Simulink are more commonly used for signal processing, and not directly for the type of class design that was needed in this case. EA, on the other hand, is made for designing class models, and it was thus selected since it can be better applied in this case.

EA is a collaborative modelling, design and management platform based on UML 2.1 and related standards. EA provides the means for creating class-diagrams, use case diagrams, sequence diagrams and many more. [52]

EA supports model validation by default. In EA, model validation compares UML models with known UML rules and with any constraints defined within the model, using the Object Constraint Language (OCL). Model validation can be run against a single UML element, a diagram or an entire package. [47]

5.3.2 Eclipse-integrated development environment

An Eclipse-integrated development environment (IDE) was selected as the environment for the implementation work. According to the survey [41], Eclipse was largely used by the industrial partners of the Evolve project. Since the implementation was done in Java and Eclipse is designed specially for developments done in this language, it suits this case well.

The Eclipse Platform is built on a mechanism for discovering, integrating, and running modules called *plug-ins*. A tool provider writes a tool as a separate plug-in that operates on files in the workspace and surfaces its tool-specific UI in the workbench. When the platform is launched, the user is presented with an IDE composed of the set of available plug-ins. [53]

The Eclipse Platform by itself is an IDE that can be used for anything in particular. The tools plugged in to the platform supply the specific capabilities that make it suitable for developing certain kinds of applications. [53] In this work, Eclipse's tool known as Java development tooling (JDT) is used, which adds Java program development capability to the platform. The JDT is included in the Eclipse Standard development kit and it supplies the means for running and debugging Java programs. A target Java virtual machine is launched as a separate process to run the Java program. [53]

5.3.3 FindBugs static analyser

Static analysis tools have been commercially available since at least 1978 (see [57], but their efficiency and usage has increased recently. Based on the SotA study [38] on the current tools and on the evaluation of different static analyser tools, FindBugs was selected as the static analyser tool for the framework. There is an eclipse plug-in available for FindBugs, which further eases its usage in this case.

FindBugs is an open source program that uses static analysis to identify hundreds of different potential errors in Java programs. It looks for instances of "bug patterns" - code instances that are likely to be errors. [58] FindBugs has been designed so that the number of false bug reports is minimised. It is stated in [58] that unlike other static analysis tools, FindBugs doesn't focus on style or formatting; specifically, it tries to find real bugs or potential performance problems.

As stated, a FindBugs Eclipse plug-in is available. The plug-in allows FindBugs to be used within the Eclipse IDE, so the code can be analysed in its implementation environment without the need to open code files in a separate program. The plug-in is also available open source.

5.3.4 Conformic Qtronic

A model-based testing tool was selected based on previous studies and knowledge on the subject. In the research [54], five different MBT tools were evaluated: the LEIRIOS test designer, Markov test logic, Conformiq Qtronic, Reactis and the Spec Explorer. In this case, the need involved modelling a data-flow oriented system, so the tool had to be data oriented (as opposed to test-control oriented, etc.). This left all tools except LEIRIOS (nowadays Smart testing [55]) and Conformiq Qtronic (CQ) [56] out of the question. Both of these two tools are suitable for use in the case study. As stated before, Eclipse was selected to be used as the IDE in the case and there is an Eclipse plug-in available for CQ. If the development and testing tools are working in the same environment, it is easier to build a less fragmented tool environment. Based on reasons related to the tool environment, the CQ was selected for use in this case study.

CQ is an Eclipse based tool for automated test generation. CQ generates software tests from high-level system models without user intervention using a UML state machine with a variation of java as the modelling language. CQ provides a wide collection of test design algorithms which can be manually selected. It also provides some scripters for widely used formats like TTCN-3 but users are able to define their own output formats as a plug-in. The tool uses a model of the system under test as a source, generates comprehensive test sets using selectable test design heuristics and writes the test sets, for example, into a database. The test sets can later be executed independently of the tool. [56]

5.4 The combined framework

In this chapter the first iteration of Evolve framework has been presented, and next it will be tested in an industrial case. The framework is formed so that it is adjustable to be suitable for different situations. In this chapter the framework has been adjusted by selecting methods, techniques and tools that are best applicable in the industrial case. As a result of these selections a one version of framework has been combined. In this section the essentials of the combined framework have been summarized.

As stated, the methods, techniques and tools for the framework used in the case have been selected and named. The selection has been made so that the framework can be used in each development phases. Table 6 lists the methods, techniques and tools in the framework for this work.

Table 6. Methods, techniques and tools in the framework for this thesis.

Phase	Methods/techniques	Tools
Requirements definition	Rapid prototypes	-
Design	Model reviews and modelling guidelines	Enterprise Architect
Implementation	Defect detection in editor	Eclipse
	Static code analysis	FindBugs
Testing and integration	Model-based testing	Conformiq Qtronic

It can be seen from the table which framework's method or technique is applied at which development phase. The table shows also the corresponding tool that will be used in applying the methods or techniques. The framework contains tools that are used in all other phases than in requirements definition phase. In that phase the V&V is done by rapid prototypes, which can be used without applying tools. In the following chapter this framework will be tested and the experiments of using each methods and tools are reported.

6. Industrial case study: the Philips case

After the framework was designed and built, it needed to be tested. Philips had a suitable real life case, in which the methods and tools presented in the previous chapter could be tested. In this chapter, the "InTouch-case" is presented. The chapter provides an introduction to the case, with a description of the Philips InTouch device followed by a step-by-step presentation of the case. At each of these steps, some of the methods in the framework were tested. The results of these experiments are presented here.

6.1 Introduction to the case

The industrial case was set up with Philips in order to try out the framework defined in the previous chapter. The objective was to explore the framework's suitability and usability in a real life scenario. The case was also planned as a learning experience on the different methods for early V&V that the framework provides. This section introduces the case and presents the device on which the application was based.

6.1.1 Case description

Philips is investigating companies' interest in sharing information in communities. In that regard, the company wanted to see a demonstration of an application that takes information from the Internet (e.g. city visiting info) and allow that information to then be presented on both mobile devices and on the device called InTouch. The idea is to connect people in other locations (looking at their mobile device), with the ones in the community that are in a single location (around the InTouch), so that they all can share the same information and discuss it.

The development work in the case was planned in three increments. The V&V framework was tested during the development of the first increment. The purpose of the first increment was to develop "InTouch-to-Computer synchronisation" (see Figure 14), and the following were required functions:

- The same web page must be visible simultaneously on the InTouch screen and on the client's computer.
- InTouch is the master device, and browsing can be done on this device exclusively.

The purpose of the second increment was to develop "InTouch-to-Phone synchronisation", in which a phone is the client for the InTouch-device. The third increment is still under development.

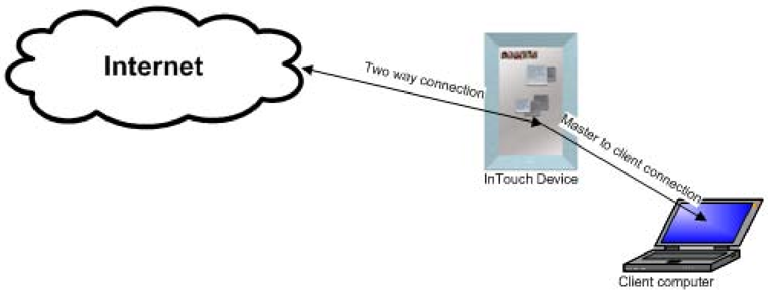


Figure 14. The first increment: InTouch-to-Computer synchronisation.

6.1.2 InTouch device

InTouch was originally developed for the Philips exhibition “Next Simplicity” in 2005. It was then presented across the world in this road show-like event during 2005 and 2006. InTouch is a mirror-like device that interacts with users. It allows them, for example, to leave notes to family members or send text messages to friends. There is a picture of the device shown in the following Figure (15). For more information, there is a presentation video available on Internet (see [59]).

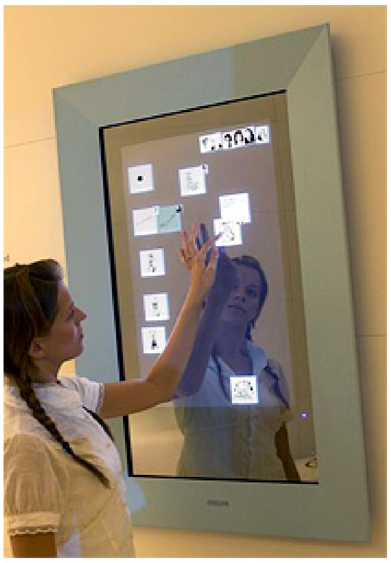


Figure 15. InTouch – an interactive mirror.

A non-disclosure agreement (NDA) was created for the case. This agreement required the application development work with the device to be confidential. This means that the issues presented in this thesis are restricted to the framework-related information and experiences (about using the methods, etc.). The developed application is introduced, but it is not described in detail. The interfaces with the InTouch are left out of the detailed discussion.

6.2 Requirements definition

The use of the framework starts at requirements definition phase as it was shown in the Table 6. During the requirements definition for the case/application, rapid prototyping was applied. A one-day session with the client was arranged for planning the requirements. First the customer presented a verbal description of the company’s expectations and explained the case’s purpose; a hands-on introduction of the device was then given. The plan for the case was based on the customer’s description and introduction and the customer was later presented with rapid prototype(s) which demonstrate the application’s behaviour.

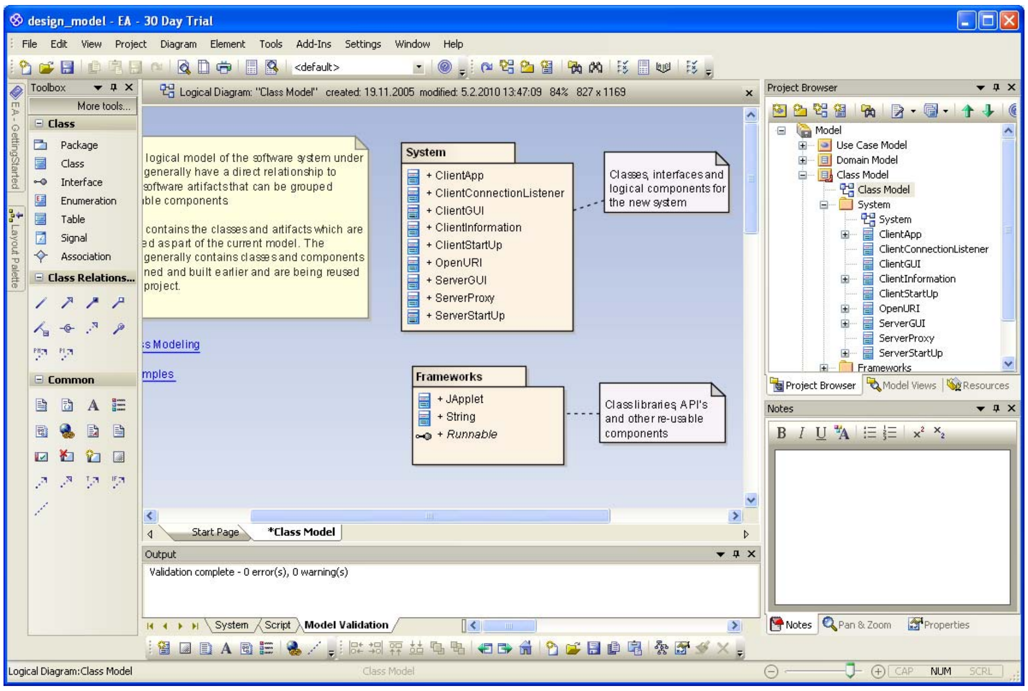
The hands-on introduction session with InTouch was successful. With the help of the technical support person, a possible approach for the case started to take form. After some planning and configurations, the prototype for the case was ready and at the end of the day, it was demonstrated to the customer. In the prototype, the InTouch was used as it is used in the case, but the actual client interaction was imaginary. This prototyping method is one form of *storyboard prototyping*.

In this case, the use of rapid prototypes was helpful. They helped in the validation and confirmed that the both sides share a similar idea of the case. The biggest help for this case was the certainty about the product to be developed is the right one – i.e. the requirements were successfully validated. The incremental development plan (see sub-section 6.1.3) was made based on the session. The plan was quickly reviewed by VTT and then reviewed and approved by the customer.

6.3 System design by modelling

A design model was created for the case to help divide the problem into smaller parts. This division helped make the structure of the software more logical and easy to follow. The framework methods was selected for the model reviews and modelling guidelines. In the end, after the model was created, it was validated with Enterprise Architect’s (EA) automatic model validation.

The design made with EA was made at the class level, and a UML diagram of the case was drawn. The application was divided into two parts: the InTouch-server side and the mobile client’s side. As mentioned earlier, the more detailed design (which could allow model checking/verification actions) is left out of this work, since the code generation was not selected. Figure 16 presents a view from the EA’s class model view. This view lists the classes that were used in the UML diagram view, which is one level deeper and shows the relations between. The “System” block in the image is the block where the classes to be implemented in this case are shown. The UML diagram model was reviewed informally during the development, and suggestions were made i.e. to add some new classes and change some relationships between existing ones. The automatic model validation was run for the resulting model.



logical model of the software system under generally have a direct relationship to software artifacts that can be grouped ble components

contains the classes and artifacts which are ed as part of the current model. The generally contains classes and components ned and built earlier and are being reused project.

Classes, interfaces and logical components for the new system

Class libraries, API's and other re-usable components

Validation complete - 0 error(s), 0 warning(s)

Figure 16. Enterprise Architect class model of the case.

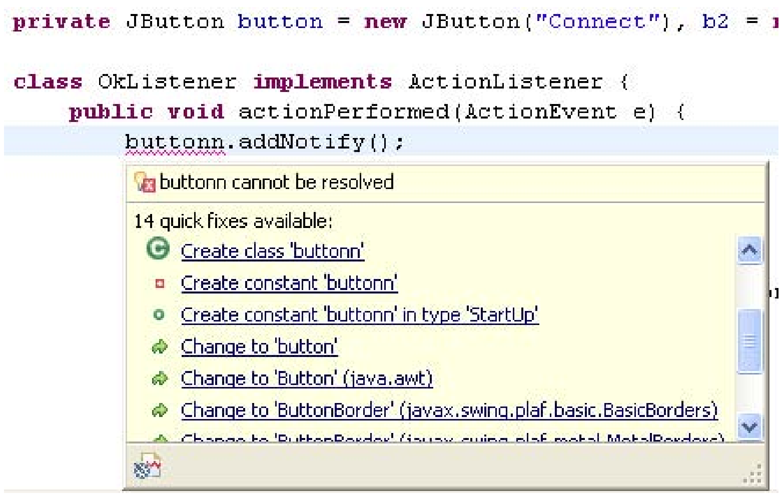
For this case, the design model, and the use of model reviews and modelling guidelines helped divide the problem to a smaller pieces. This can be thought of as defect prevention activity, since it eases the implementation work and increases software's understandability and readability in later phases. Informal review sessions were suitable (because of their lightness) and useful for this case. The reviews provided good ideas for improving the application class structure. Different approaches to the problem were evaluated and the most adequate one was selected. Automatic model validation did not provide any additional information for this case, since validation was completed with no errors or warnings throughout the model development. It is possible that the use of this technique would require a larger and more detailed level model to be more useful. All in all, the architectural design model was successful, and the rest of the work was done according to this model.

6.4 Software implementation

Since the InTouch device was not available during the development, the work was made so that the InTouch device was simulated by a Windows desktop PC. The application software was implemented in Java, using Eclipse IDE's Java development tooling plug-in. The early V&V methods applied during this phase were defect detection in editor and static code analysis. The following sub-sections describe how well these methods worked in this case.

6.4.1 Defect detection in editor

Eclipse does (automatic) defect detection every time the project is saved (and the project is built). Here, defect detection is done by highlighting and indicating the errors and warnings from the source code. The indication of these errors is made by underlining the defective part on the code (see Figure 17). There is also a separate “Problems” window which lists all active errors and warnings found in the project. Eclipse also offers “quick fix” suggestions for the problems. For example, Eclipse can generate the classes or add unimplemented methods into the project. Thus the “quick fix” can be considered a code generation of some type.



```
private JButton button = new JButton("Connect"), b2 =

class OkListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        buttonn.addNotify();
```

buttonn cannot be resolved

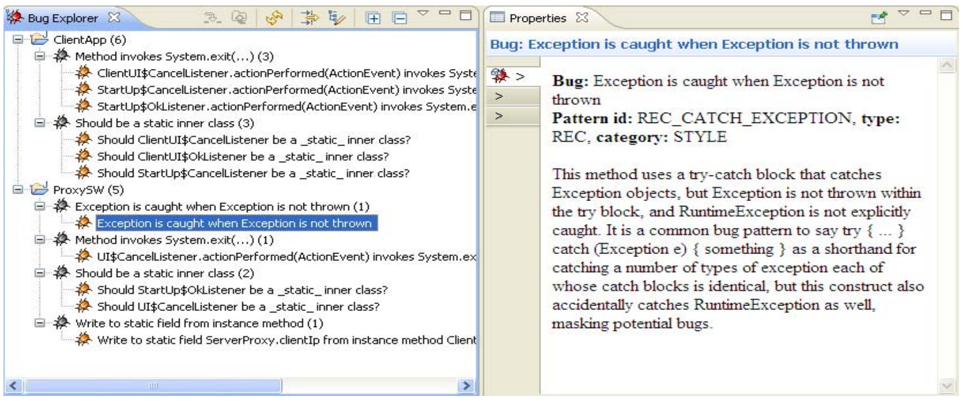
14 quick fixes available:
- Create class 'buttonn'
- Create constant 'buttonn'
- Create constant 'buttonn' in type 'StartUp'
- Change to 'button'
- Change to 'Button' (java.awt)
- Change to 'ButtonBorder' (javax.swing.plaf.basic.BasicBorders)

Figure 17. Defect detection and quick fix suggestions in Eclipse.

In this case, defect detection in the editor worked well. The compilation-time-defects were sorted out from the code soon after they were made, and in the end, the application was ready to be run without the need for debugging during the compilation. Since Eclipse also reported the compilation warnings (and offered a quick fix for those), it was easy to accomplish tasks like removing unused libraries from the import statements. Getting rid of warnings can for example decrease the size of the application and make it run faster. During this case, defect detection in the editor helped in the early removal of the defects from the implementation.

6.4.2 Static code analysis

The selected tool for the static code analysis was FindBugs (see section 5.3). The FindBugs Eclipse plug-in was installed so the analysis could be made in the same environment as the implementation work. Once the (manually executed) scan for the source code is made, FindBugs reports the found problems and marks those in the source code with tags. There is also a window available that lists all problems found. In this window, users can see an explanation of the bug (see Figure 18) and some other information, for example bug’s priority and resource.



Bug Explorer

```
ClientApp (6)
    Method invokes System.exit(...) (3)
        ClientUI$CancelListener.actionPerformed(ActionEvent) invokes Syste
        StartUp$CancelListener.actionPerformed(ActionEvent) invokes Syste
        StartUp$OkListener.actionPerformed(ActionEvent) invokes System.e
    Should be a static inner class (3)
        Should ClientUI$CancelListener be a _static_ inner class?
        Should ClientUI$OkListener be a _static_ inner class?
        Should StartUp$CancelListener be a _static_ inner class?
ProxySW (5)
    Exception is caught when Exception is not thrown (1)
        Exception is caught when Exception is not thrown
    Method invokes System.exit(...) (1)
        UI$CancelListener.actionPerformed(ActionEvent) invokes System.ex
    Should be a static inner class (2)
        Should StartUp$OkListener be a _static_ inner class?
        Should UI$CancelListener be a _static_ inner class?
    Write to static field from instance method (1)
        Write to static field ServerProxy.clientIp from instance method Client
```

Properties

Bug: Exception is caught when Exception is not thrown

Bug: Exception is caught when Exception is not thrown
Pattern id: REC_CATCH_EXCEPTION, **type:** REC, **category:** STYLE

This method uses a try-catch block that catches Exception objects, but Exception is not thrown within the try block, and RuntimeException is not explicitly caught. It is a common bug pattern to say try { ... } catch (Exception e) { something } as a shorthand for catching a number of types of exception each of whose catch blocks is identical, but this construct also accidentally catches RuntimeException as well, masking potential bugs.

Figure 18. FindBugs defect description window.

In this case, FindBugs made both correct and false reports of bugs. Some of the found (correct) problems seem like additional warnings that the defect detection in editor reports (and were not so crucial), but potential causes for run-time defects were also found. In this case, the clear defect description was helpful in sorting out the correctly identified bugs from the false reports. In addition, the description led to suggestions for appropriate and applicable corrections in this case. The Eclipse plug-in made the tool work faster and this encouraged to analyse the source code more often. In this case, static code analysis helped to sort out few possible run-time defects that would be hard to find in the later phases of the development.

6.5 System level testing

The system level testing was done using model-based testing (MBT). For the test model, Conformiq Qtronic was used right after the requirements for the case had been defined. The test model designed for the case was done according to a typical use case of the system. This model was then used to generate several different test cases.

The MBT process was divided in the five phases, as presented in the previous sub-section (6.2.5). The first task involved modelling the SUT and/or its environment. The following figure (19) illustrates the test setup that was used for this case. The InTouch server is the system under test (SUT) and its interfaces are simulated by the test platform. The InTouch server is running in its normal operational mode, so it is the same as a real life situation from its perspective (it doesn't recognise that the messages are simulated). The test platform sends the specified messages to InTouch using some of the interfaces and then expects the (appropriate) answer either in the same interface or in some other interface, depending on the situation. By comparing the sent and received values, the test platform will create the test report, which can be analysed to see that the system works as expected.

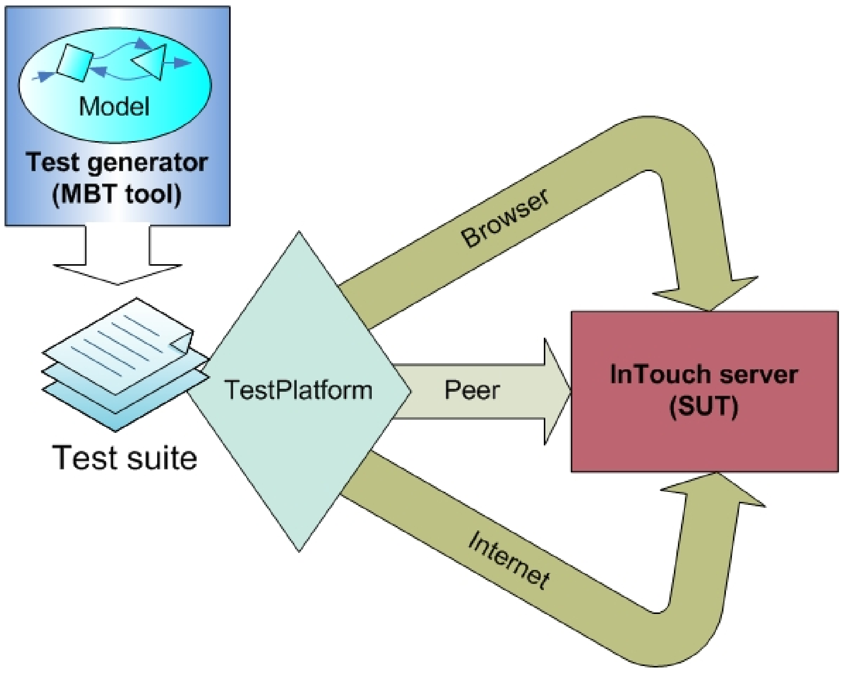


Figure 19. Simulated test interfaces.

The actual test modelling could be started after the test interfaces had been defined. The test model for the system is shown in Figure 20. This model is drawn with Conformiq Qtronic and it is based on one basic use case of the system. In the use case, the user clicks on a link in the browser of InTouch, and the same web page is shared with the client device. The model allows different paths to be taken while creating tests, thus allowing different kinds of test cases to be generated.

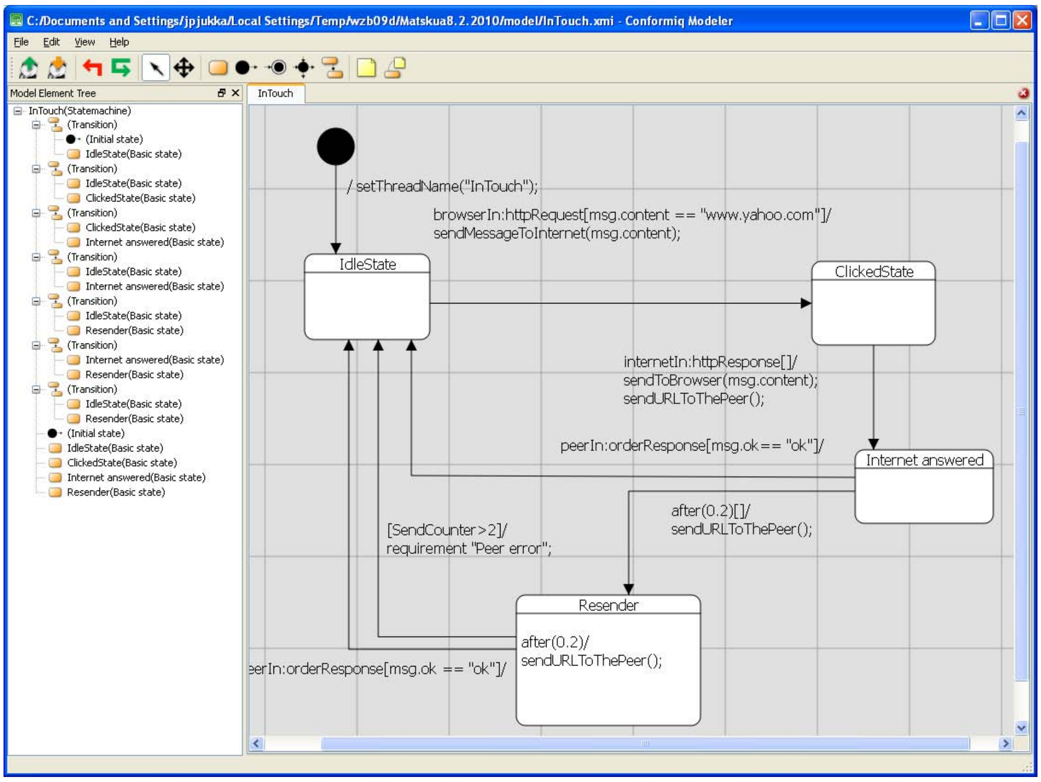
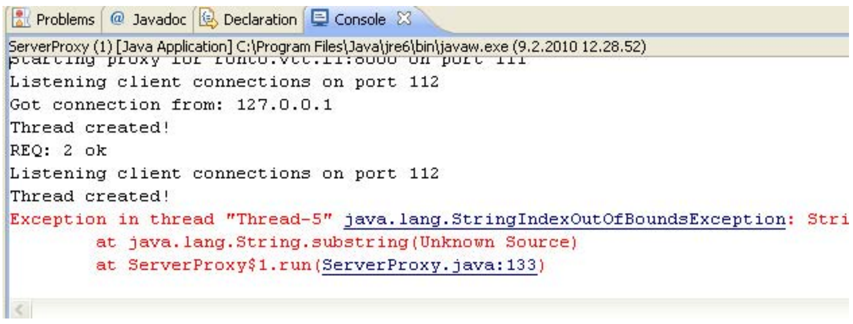

C:/Documents and Settings/jpjukka/Local Settings/Temp/wzb09d/Matskua8.2.2010/model/InTouch.xmi - Conformiq Modeler
File Edit View Help
Model Element Tree
InTouch
InTouch(Statemachine)
(Transition)
(Initial state)
IdleState(Basic state)
ClickedState(Basic state)
Internet answered(Basic state)
Resender(Basic state)
/ setThreadName("InTouch");
browserIn:httpRequest[msg.content == "www.yahoo.com"]/
sendMessageToInternet(msg.content);
IdleState
ClickedState
internetIn:httpResponse[]/
sendToBrowser(msg.content);
sendURLToThePeer();
peerIn:orderResponse[msg.ok== "ok"]/
Internet answered
after(0.2)[]/
sendURLToThePeer();
[SendCounter>2]/
requirement "Peer error";
Resender
after(0.2)/
sendURLToThePeer();
eerIn:orderResponse[msg.ok == "ok"]/

Figure 20. The Conformiq Modeler test model.

The test model above was used to generate the test cases for the system. The generated test cases were stored in XML form, which makes them easy to parse and use on the test platform. In this case, a total of three different test cases (scenarios) were generated for the system. Both positive and negative scenarios were generated with the amount of test steps ranging from six to 280. This was done in order to test the possibilities of the method/tool. Only the positive test case of normal operations had to be executed for this (demonstration) case's purposes.

The next thing to do was to create the test platform that is used to actually run the generated tests. The test platform reads the test data from the generated test cases, and uses that information to control the system under test through the simulated interfaces. The test platform was written in Java and it contained two parts. The first addressed interfacing: a short piece of software that runs three threads for listening to the interface ports and writes incoming activity into test report. The second part was the control software. The control software parses the XML file, and by using the test steps found there, proceeds with the test case step by step. The fields under the test step contain data like the port number or the value of the data to be sent.

After the tests had been finally created, the test case could be executed. The test platform worked well right away, and the test case was successfully run. During the first run of the case, a run-time defect from the software was uncovered (see Figure 21), and the test case was aborted. The bug was fixed and the case was re-executed. No more errors occurred after this and an analysable test report was created. After checking the data from the report to make sure that everything was correct, the case could be marked as passed.



```
ServerProxy (1) [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (9.2.2010 12.28.52)
Starting proxy for ronco.vtt.fi:8000 on port 111
Listening client connections on port 112
Got connection from: 127.0.0.1
Thread created!
REQ: 2 ok
Listening client connections on port 112
Thread created!
Exception in thread "Thread-5" java.lang.StringIndexOutOfBoundsException: Stri
        at java.lang.String.substring(Unknown Source)
        at ServerProxy$1.run(ServerProxy.java:133)
```

Figure 21. Run-time error found during the test case.

MBT was successfully tested in this case. Based on this test, it is a good addition to the Evolve framework.

In this case, the most time-consuming work in MBT was the creation of the test model and the test platform. It should be noted, though, that after these have been created, multiple different test cases can be generated. Especially in cases in which the logic implementation of the test platform is made to be general (as in this work, XML-file fully guides its execution), the same test platform can be used to run many different test cases, even those from different models (albeit with small modifications).

The test case generation worked quite well in this case. The generated cases required some manual configuring though, before they were executable. This configuration was needed for example for setting the correct interface ports and because it was not clear how this could be done automatically with the tool.

Selecting XML-based test scripts was a good solution for this case. The tool would have provided the means to generate executable Java programs, which could enable the whole test platform to be generated. However, in this case, it was seen to be easier to implement a separate test platform and generate the test cases as XML files for several reasons e.g. since there were so many different interfaces involved.

For this case, MBT was a particularly heavy method to be used. Based on this experiment, MBT would prove even more useful if the size of the project increases. It seems that quite a lot of effort is needed for designing and creating the test models and platforms, but after these are successfully completed, MBT allows an extensive set of (both positive and negative) test cases to be generated for the use case. Based on this case, MBT may help also in test automation by creating executable scripts. For example, test cases for stress testing and testing with different input and output values could be easily added with the help of the MBT.

7. Discussion

The results of this work have already been presented in this report, but do these results satisfy the objectives set for this thesis? This chapter discusses how well the objectives of this thesis have been met. These objectives, discussed here, include both the given (in the introduction) and defined (framework) requirements for the work. This discussion goes through both the background research as well as the assembly of the framework used for the testing results. After this some similar research studies are presented and compared to this thesis. Finally, the chapter concludes with a discussion about the possibilities for work in the future i.e. how research will continue and how the framework will be built further on.

7.1 Background study on V&V

The area of V&V is diverse and widely spread. The first major challenge during this thesis was to study the theory behind the V&V process and its main activities. No single source was found that would list these activities or could be used throughout the study. So, to be able to carry out this research, a study on multiple sources, including books, articles and online materials, was conducted.

Obviously, it was a challenging task to fit all of this information in this thesis without leaving out any essentials and while keeping to the subject. Model-based development and test automation were left out of this background study, since these areas are so large that they could easily form a thesis of their own. However, it can be said that the outcome, as presented in this thesis, is extensive. The main V&V activities were found and their essentials were presented as well as the guidelines for planning. It can also be said that this information is consistent, considering the quantity of sources used. The conducted background study on V&V created a good starting point for further research on this thesis and forms an extensive set of information on basic V&V activities that can well be used as a reference for other studies in this area.

7.2 Industrial survey on state-of-the-practice

One of the objectives of this thesis was to conduct a survey on the V&V process in the industry. This survey was completed successfully, and an online answer form was generated for responses. The industrial partners of the Evolve project were asked to fill in the form, and a sufficient amount of responses was received.

The results of the survey were analysed and the analysis report was put together as part of the work related to this thesis. This work was reviewed by the Evolve project's industrial partners, who accepted it as a deliverable for the project. The process of publishing the survey results as a VTT publication has been already started. This publication will provide valuable data on the current state-of-the-practice of V&V in software development in the European setting.

The questions in the survey were planned carefully to make the results of (all parts of) the survey usable for guiding this thesis and later, for the Evolve project. In this regard, the results provided important information about the partner companies' development environments and practices. This information was used to select the tools that were used in the case study. The results were also used to define the requirements for the Evolve framework. All in all, the survey was successful and provided (at least) the inputs for the case that it was supposed to provide.

7.3 Framework assembly

The final goal of this thesis work was to create a first iteration of the Evolve early V&V framework. Based on previous research done as part of this thesis, the framework was succesfully assembled. In addition to the issues presented earlier in this chapter, this research included the state-of-the-art (SotA) study on early V&V methods and techniques. The previous research was used to define the requirements for the Evolve framework and to then create the conceptual level model of the framework.

A SotA study was done as part of this thesis. The main focus of the study was to find methods and techniques for applying early V&V in different phases of product development. An extensive SotA study was successfully conducted, and it was done as a deliverable for the Evolve project. The possibility of publishing the results of the study in some form is also being considered. Some novel methods and techniques were found during the study, and these findings were used in this thesis to define the structure for the Evolve framework. SotA gives the guidelines and descriptions on using the methods and techniques of the framework.

Finding and collecting the requirements for the Evolve framework was one of the objectives of this thesis. Based on research done as part of this thesis, the requirements were defined. Those that were supposed to be tackled during this thesis were selected and sorted out from other requirements. These selected requirements were used when planning how to form the Evolve framework to select the most appropriate methods and techniques, which were tested in this work.

The formed framework is a toolbox-like combination of methods and techniques for different phases of product development lifecycle. This framework is adaptable for different companies and for different situations inside companies. The selection of methods and tools used in this case was done well, and they form a solid system for applying the early V&V in every phase of the development in this case.

The form of the Evolve framework is not final in its current form. Many requirements were left open after this thesis and more requirements will probably be added during the progress with research. Some requirements for this work (e.g. R001) will probably be developed further after this work as well. The issues for the future will be discussed later in this chapter. Nevertheless, the framework in its current form will serve as the backbone for the forthcoming research and implementation work done during the Evolve project.

7.4 Industrial case

A real-life industrial case was done with Philips in order to see how the Evolve framework works in practise and determine how it helps in early defect detection and prevention. The usability and functionality of the framework's methods and techniques were tested at the different stages of the product development. In this thesis, the tests and findings done as part of this industrial case were reported.

The development work in the case was made under an NDA with the customer. This agreement limited our reporting on the developed application but this was not a problem since the interest in this work was on the development process of the application and not in the application itself. As for the application, it was demonstrated to the customer successfully. The customer has the right to demonstrate the application further to Philips' own customers.

The case was educational and the objectives concerning the framework's exploration were achieved. New early verification and validation methods were tried out, and valuable information was garnered on their usage and effectiveness in a real-life case. In addition, knowledge and hands-on experience with using tools for different phases of the product development were also obtained.

The case proved that the framework, which was assembled in this thesis, is usable in a real life setting. The framework helped to detect some defects at earlier phases and prevented them in the later stages. The time needed for the V&V actions was quite large in this case, but it was partly due to the fact new tools and methods had to be learned. The integrated tool environment (like Eclipse integrated with the V&V plug-ins), proved very helpful in this case. The time spent with the required V&V actions is shortened with the help of these integrations. It was not possible to compare traditional development (without the use of early V&V framework), but it can be said that, at least in this case, the early V&V Evolve framework was worth using and helpful in early defect detection and prevention.

7.5 Comparison against similar work

During this work multiple researches in the area of the early V&V have been gone through. Many of these researches conclude that applying early V&V is highly recommended, but these do not usually provide any concrete means for doing this. There were not so many researches found, which actually present (more than one) solutions for early V&V.

In addition to this thesis, [40] is a research that suggests some solutions for improving early V&V. This research was used as a reference for this thesis when the lifecycle model was defined for the Evolve framework. However, the solutions provided in [40] are more in the project management level than in the product development work level. The research suggests solutions based on the historical project data like defect classification or defect causal analysis. These solutions are more like solutions that can be applied when planning early V&V activities and, unlike the solutions provided in this thesis, they are not the kind of solutions which would uncover new defects during the actual product development. The management viewpoint is covered also in [31]. It has been planned, that the V&V management and planning issues will be considered also in the next iteration of the Evolve framework.

Also other researches have been made, that contain some single methods or techniques to be used in early V&V of some phase of the development. Many of these have been gone through and some have been selected to be attached in the Evolve framework.

7.6 Future work

The first iteration of Evolve’s early V&V framework has been defined and presented in this thesis work. In the future, the related research and development work will be continued; during this time, the framework may change and new requirements for it may be defined. This section presents some ideas about the framework’s future development as well as some of its exploitation possibilities. Finally, at the end of this section, the results are summarised and plans for continuing the work related to this thesis are presented.

At first, solutions will be planned and implemented to continue with the framework development; these solutions will fulfil the rest of the requirements defined for the framework in this thesis. For example, planning should be done on how to better guide the selections of V&V methods and techniques to be applied in different situations. There could be, for instance, some kind of tool inside the framework which could be used to guide these selections throughout the product development lifecycle. This tool could also help, for example, when planning the V&V process or to allow for progress reports. The requirements table (Table 5) also defines many other requirements that are still on the future work task list.

In this thesis, some of the methods and techniques inside the framework were tried out in a real-life application development. However, there are still some (less known) methods in the framework, which should be tested before they can be recommended for use in the industry. As this work was done in traditional coding style, it would also be interesting to test out how well the framework works in model-driven development concept.

During the case study of this work, it was noted that the V&V tools were well integrated within the development environment, making them easier to use. This was so helpful that it is necessary to plan how the framework could be developed to support these kinds of integrations and allow more tools to be integrated. For example, there could be some kind of integrations or tool chains to bring together different methods of the framework.

As stated before, the results of this research are already being applied. Two project deliverables have been already created, and the process for publishing a VTT working paper about the survey results has begun. The possibility of releasing the SotA report as a VTT publication or as a white paper is also under consideration. In addition, the goal is to write (at least) one conference paper based on the research done as part of this thesis.

8. Conclusions

In this chapter, the contents and the most notable results of this thesis work are summarised. First, there is a short revision on the thesis objectives, followed by a summary of the contents and results. The chapter concludes with a brief consideration of how this thesis work succeeded overall.

This thesis work was done as a part of the Evolve project, and the thesis results will be used in the project's future actions. The basic concept behind the Evolve project is to improve the possibilities to apply early V&V to products in the industrial setting. The objectives for this thesis were based on this same concept.

To approach the problems described above, some smaller, intermediate goals for the work were specified. The first objective was to design and conduct an industrial survey among the Evolve project partners. The survey was drawn up in order to get information about the state-of-the-practice of V&V in the industry. The next goal was to define the requirements for the Evolve framework. Finally, after the requirements had been defined, the objective was to design and build the Evolve framework that would help companies apply V&V activities and methods earlier in their development work.

This thesis work was done in a structured, step-by-step manner and the main results were obtained during each step.

The work began with a theoretical background study on the main V&V activities applied in product development. This background study was extensive and provided a compact set of information that can be used as reference for other research projects. This set is an important addition to the theory on this subject, since no other sources were found that list this information so completely.

In the next phases, two large studies were carried out. The first was the industrial survey for the Evolve project partners and the second was the literature study on state-of-the-Art (SotA) of early V&V methods and techniques. Both of these studies were done as work related to this thesis and fulfilled their purpose well.

The survey provided a valuable set of results regarding the current state-of-the-practice of V&V in software development in the European setting. This set was used in the later phases of this work, for example in requirements definition. The results have been also published as a deliverable of the Evolve project and the process of making them available as a VTT publication has been started.

The SotA study revealed various methods and techniques in scientific literature for applying early V&V. These results were used later when the Evolve framework was formed. Also these results have been published as an Evolve deliverable and are being considered for public release.

Based on our previous research, the requirements were specified for the Evolve framework. Out of these requirements, the ones for this initial version were identified. These requirements were incorporated once the framework was finally designed and built. The resulting framework is a combination of methods and techniques to be applied during the different phases of the development work. The methods and techniques included were identified during the SotA study.

In the end, the framework was tested in an industrial real world use case, which proved that the framework is usable. The case also provided valuable data and knowledge on using different V&V methods and techniques during the early phases of the process.

Much research work was required along with a background study to build the first iteration of Evolve's early V&V framework. Many objectives were covered and a different set of results was provided. Now that the work is complete, the outcome has proved to be surprisingly consistent and organised given the enormous scope of the subject. The resulting framework is not final, but the work done as part of this Master's thesis forms a solid groundwork for future research and development in this area.

References

[1] Boehm B. (1987) Industrial software metrics top ten list. IEEE Software, Vol 4(5), pp. 84–85.

[2] Humphrey W. (2002) The Payoff from Software Quality. Computerworld,
URL: http://www.computerworld.com/developmenttopics/development/story/0,10801,71222,00.html.

[3] Boehm B., & Basili V. R. (2001) Software defect reduction top 10 list. Los Alamitos, CA, USA, IEEE Computer Society Press, vol 34(1), pp. 135–137.

[4] IEEE standard for software verification and validation (1998).

[5] Andriole S. (1986) Software Validation, Verification, Testing, and Documentation, Petrocelli Books, Princeton, NJ, 480 p.

[6] (Read 12.5.2009) Tran E. Verification/Validation/Certification.,
URL:http://www.ece.cmu.edu/~koopman/des_s99/verification/.

[7] (Read 20.5.2009) V-Model definition.
URL: http://en.wikipedia.org/wiki/V-Model_(software_development).

[8] (Read 3.6.2009) Regional ITS Architecture Guidance Document, Section 7.3.
URL: http://ops.fhwa.dot.gov/publications/regitsarchguide/.

[9] Toroi T. (2005) Tietojärjestelmän testaus, Lecture, Kuopion yliopisto.

[10] (Read 12.6.2009) The definition for Rational Unified Process. URL: http://en.wikipedia.org/wiki/RUP.

[11] Pressman R. (2009) Software Engineering: A Practitioner’s Approach. 7th edition, Boston, McGraw-Hill, 928 p.

[12] Jacobson I., Booch G. & Rumbaugh J. (1999) The Unified Software Development Process, Addison Wesley Longman Inc, 463 p.

[13] Hardgrave B. & Wilson L. (1994) An Investigation of Guidelines for Selecting a Prototyping Strategy, Journal of Systems Management, Vol 45(4), pp. 28–35.

[14] (read 1.6.2009) Prototyping online article: What is Prototyping?.
URL: http://www.umsl.edu/~sauterv/analysis/prototyping/proto.html.

[15] Kotonya G. & Sommerville I. (1997) Requirements Engineering: Process and Techniques. John Wiley & Sons, 294 p.

[16] Parviainen P., Hulkko H., Kääriäinen J., Takalo J. & Tihinen M. (2003) Requirements engineering, Inventory of technologies, VTT Publications 508.

[17] (Read 12.6.2009) Smith, R. Simulation article. Encyclopedia of Computer Science,
URL: http://www.modelbenders.com/encyclopedia/.

[18] Jaakola M., Parviainen P. & Kanstrén T. (2007) Simulation Methods and Tools. TWINS project deliverable, VTT.

[19] Law A. (2006) Simulation Modelling & Analysis. 4th Editition, McGrawhill, 784 p.

[20] Schmid R., Ryser J., Berner S, Glinz M., Reutemann R & Fahr E. (2000) A Survey of Simulation Tools for Requirements Engineering. Special Interest Group on Requirements Engineering, German Informatics society.

[21] IEEE Standard for Software Reviews and Audits (2008).

[22] IEEE Standards Collection, Software Engineering (1994).

[23] Parviainen P. (1996) ProMETRI project REVIEW TYPES – literature study. VTT.

[24] (Read 30.6.2009) Five Types of Review. URL: http://smartbear.com/docs/ book/code-review-types.pdf.

[25] (Read five12.10.2009) The definition for software testing.
URL: http://en.wikipedia.org/wiki/Software_testing.

[26] Kärki M. (2001) Testing of object-oriented software – Utilisation of the UML in testing. Master's Thesis. University of Oulu, Department of Electrical Engineering, Oulu.

[27] Black R. (2002) Managing the Testing Process - Practical Tools and Techniques for Managing Hardware and Software Testing. 2nd Edition, John Wiley & Sons, 528 p.

[28] (Read 1.7.2009) Software Testing Techniques.
URL: http://www.his.sunderland.ac.uk/~cs0mel/comm83wk5.doc.

[29] Beizer B. (1992) Software testing techniques. 2nd edition, New York, NY, Van Nostrand Reinhold, 550 p.

[30] (Read 15.7.2009) An online information resource for software testers, URL: http://www.testingfaqs.org/.

[31] Belt P. (2009) Improving verification and validation activities in ICT companies – product development management approach. University of Oulu.

[32] (Read 15.9.2009) Risk Impact/Probability Chart, Available online.
URL: http://www.mindtools.com/pages/article/newPPM_78.htm.

[33] (Read 17.9.2009) Software Testing Standard. URL: http://www.testingstandards.co.uk/.

[34] (Read 17.9.2009) Glossary of Software Testing Term. URL: http://www.testingstandards.co.uk/bs_7925-1_online.htm.

[35] Evolve project Formal Project Plan (2007).

[36] Harrold M. (2000) Testing: A Roadmap. Conference on Software Engineering, Limerick, Ireland, pp. 61–72.

[37] Wallace D. & Fujii R. (1989) Software Verification and Validation: An Overview. IEEE Software.

[38] Pesola J.P., Bendas D., Heinonen S., Tanner H. & Teppola S. (2009) State-of-The-Art study on Processes, Methods, Techniques and Tools for Applying Early V&V, Evolve project deliverable, VTT.

[39] Fisher M. (2007) Software Verification and Validation. An Engineering and Scientific Approach. Springer.

[40] van Moll J., Jacobs J. Freimut B. & Triekens J. (2002) The Importance of Life Cycle Modelling to Defect Detection and Prevention, IEEE 2002.

[41] Pesola J.P. & Tanner H. (2009) Survey on State-of-The-Practice of V&V in the Industrial Setting, Evolve project deliverable, VTT.

[42] Runeson P., Andersson C. & Höst M. (2003) Test Processes in Software Product Evolution – A Qualitative Survey on the State of Practice. JSME'03 – Journal of Software Maintenance and Evolution, Vol. 15(1), pp. 41–59.

[43] Wojcicki M.A. & Strooper P. (2006) A State-of-Practice Questionnaire on Verification and Validation for Concurrent Programs. Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD-IV), Portland, Maine, USA.

[44] Andersson C. & Runeson P. (2002) Verification and Validation in Industry – A Qualitative Survey on the State of Practice. International Symposium on Empirical Software Engineering (ISESE'02), Nara, Japan.

[45] Lobo L. & Arhur J. (2009) Effective Requirements Generation: Synchronizing Early Verification & Validation, Methods and Method Selection Criteria. Department of Computer Science, Virginia Tech Blacksburg, Virginia.

[46] Kelly S. & Tolvanen J.P. (2008) Domain-Specific Modelling – Enabling Full Code Generation. IEEE Computer Society, John Wiley & Sons, 427 p.

[47] den Haan J. (read 8.11.2009) MDE - Model Driven Engineering – reference guide.
URL: http://www.theenterprisearchitect.eu/archive/2009/01/15/mde---model-driven-engineering----reference-guide.

[48] Louridas P. (2006) Static code analysis. IEEE Software, pp. 58–61.

[49] (Read 8.11.2009) DACS (The Data & Analysis Center for Software) Gold Practices Website. URL: http://www.goldpractices.com/practices/mbt/.

[50] Utting M. & Legeard B. (2007) Practical Model-Based Testing, A tools approach. Elsevier Inc. USA, 433 p.

[51] Blackburn M., Busser R. & Nauman A. (2004) Why Model-Based Test Automation is Different and What You Should Know to Get Started. International Conference on Practical Software Quality and Testing. (PSQT/PSTT).

[52] (Read 10.12.2009) Enterprise Architect software design tool, URL: http://www.sparxsystems.com/.

[53] (Read 12.12.2009) Eclipse platform technical overview (2003).
URL: http://www.eclipse.org/whitepapers/eclipse-overview.pdf.

[54] Puolitaival O-P. (2007) Adapting model-based testing to agile context. Master's Thesis. University of Oulu, Department of Electrical and Information Engineering, Oulu.

[55] (Read 20.12.2010) Smart testing (formerly LEIRIOS test designer) model-based testing tool, URL: http://www.smarttesting.com.

[56] (Read 20.12.2010) Conformiq Qtronic model based testing tool,
URL: http://www.conformiq.com/qtronic.php.

[57] Johnson S. (1978) Lint, a C program checker. No. Comp. Sci. Tech. Rep 65, Bell Laboratories.

[58] (Read 6.1.2010) Grindstaff C. FindBugs, Part 1: Improve the quality of your code, IBM, URL: http://www.ibm.com/developerworks/java/library/j-findbug1/.

[59] (Read 12.12.2009) Presentation video on Philips InTouch device.
URL: http://www.dailymotion.com/video/x6hi1d_philips-intouch-1_tech.

Series title, number and report code of publication

VTT Publications 736
VTT-PUBS-736

Author(s)
Jukka-Pekka Pesola

Title

Building Framework for Early Product Verification and Validation

Abstract

This thesis report presents a step-by-step approach on verifying and validating products early during the development. The final result is a framework that helps detect defects during the early phases of the product lifecycle and prevent them during the later phases.

First, this report presents extensive research on the main verification and validation activities. These activities – simulation, prototyping, reviews and testing – are presented in the report, along with some suggestions for planning them effectively.

The challenges for early product verification and validation were identified and researched by conducting two large studies. The first was a survey that was conducted to gain insight on the current state of the early verification and validation process in the industrial environment. The survey resulted in valuable set of results regarding the current state-of-the-practice of verification and validation in software development in the European setting.

The second research was a literature study on state-of-the-art of early verification and validation processes, methods and techniques. The study resulted in exhaustive collection of methods and techniques that can be applied for product early verification and validation in the different phases of the project. The main results of both of these studies are presented in this thesis.

Based on the results of the studies mentioned above, the requirements for the framework were defined. As a result, 16 requirements for the framework were specified and these are described in this work.

The design and building of the framework according to the requirements then started. The results, including a set of methods and techniques for applying early verification and validation in product development, are presented in this thesis. Finally, the framework was tested in a real-life industrial case and the experiments and results from this case are presented.

ISBN 978-951-38-7393-6 (URL: http://www.vtt.fi/publications/index.jsp)		
Series title and ISSN VTT Publications 1455-0849 (URL: http://www.vtt.fi/publications/index.jsp)		Project number 24506
Date April 2010	Language English, Finnish abstr.	Pages 75 p.
Name of project EVOLVE		Commissioned by VTT
Keywords Validation, verification, testing, V&V, product lifecycle		Publisher VTT Technical Research Centre of Finland P. O. Box 1000, FI-02044 VTT, Finland Phone internat. +358 20 722 4520 Fax +358 20 722 4374

Julkaisun sarja, numero ja raporttikoodi

VTT Publications 736
VTT-PUBS-736

Tekijä(t)
Jukka-Pekka Pesola

Nimeke

Kehikko tuotteen aikaiseen verifiointiin ja validointiin

Tiivistelmä

Tässä työssä etsitään ratkaisuja tuotteen verifioinnin ja validoinnin aikaistamiseksi. Työssä käydään läpi vaiheittainen lähestymistapa ongelmaan. Työn lopputuloksena määritellään kehikko, joka auttaa löytämään ohjelmistovikoja jo tuotteen elinkaaren aikaisissa vaiheissa ja estämään niiden siirtymistä myöhempiin vaiheisiin.

Työn aluksi esitetään kattava tutkimus tärkeimmistä verifiointiin ja validointiin käytetyistä toiminnoista. Nämä toiminnot – simulointi, protypointi, katselmoinnit ja testaus – esitellään ja ohjeita niiden suunnittelemiseksi annetaan.

Tuotteen aikaisen verifioinnin ja validoinnin haasteita kerättiin ja tutkittiin tekemällä kaksi laajaa tutkimusta. Näistä ensimmäinen oli teollinen kysely, jonka tarkoitus oli antaa näkemystä aikaisesta verifioinnista ja validoinnista teollisessa ympäristössä. Kyselyn tuloksena saatiin hyödyllinen tietopaketti verifioinnin ja validoinnin käytännöistä Euroopan tason ohjelmistokehityksessä.

Toinen tutkimus oli kirjallinen selvitys nykyisistä verifioinnin ja validoinnin prosesseista, metodeista ja tekniikoista. Tutkimuksessa koottiin yhteen laaja kokoelma metodeja ja tekniikoita, joita voidaan käyttää tuotteen aikaiseen verifiointiin ja validointiin projektin eri vaiheissa. Molempien tutkimusten päätulokset on esitelty tässä työssä.

Kehikolle määritettiin vaatimukset yllämainittuihin tutkimustuloksiin perustuen. Vaatimusmäärittelyn tuloksena saatiin kaikkiaan kuusitoista vaatimusta, jotka esitellään tässä työssä.

Tämän jälkeen kehikon suunnittelu ja rakentaminen vaatimusten mukaiseksi alkoi. Tuloksesksi saatiin kehikko, joka sisältää metodeja ja tekniikoita aikaiseen verifiointiin ja validointiin tuotekehityksessä. Lopuksi kehikkoa kokeiltiin oikeassa teollisessa tapauksessa. Tapauksesta saatiin tuloksena kokemuksia, jotka raportoidaan tässä työssä.

ISBN
978-951-38-7393-6 (URL: http://www.vtt.fi/publications/index.jsp)

Avainnimeke ja ISSN VTT Publications 1455-0849 (URL: http://www.vtt.fi/publications/index.jsp)	Projektinumero 24506

Julkaisuaika Huhtikuu 2010	Kieli Englanti, suom. tiiv.	Sivuja 75 s.

Projektin nimi EVOLVE	Toimeksiantaja(t) VTT
Avainsanat Validation, verification, testing, V&V, product lifecycle	Julkaisija VTT PL 1000, 02044 VTT Puh. 020 722 4520 Faksi 020 722 4374

ISBN 978-951-38-7393-6 (URL: http://www.vtt.fi/publications/index.jsp)
ISSN 1235-0621 (soft back ed.)