A software testing approach supported by a tool environment for the development of component tests

Hermans, John

Award date:
2006

TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computing Science

MASTER'S THESIS

A Software Testing Approach Supported by a Tool Environment for the Development of Component Tests

John Hermans
j.g.f.hermans@student.tue.nl

Supervisor: Dr. J. M. T. Romijn
jromijn@win.tue.nl
Technische Universiteit Eindhoven, Eindhoven, The Netherlands.

Tutor: Ir. G. Zwartjes
gzwartjes@research.intersoft.nl
Intersoft Software Reseach, Eindhoven, The Netherlands.

Eindhoven, March 2006

A Software Testing Approach Supported by a Tool Environment for the Development of Component Tests

John Hermans

April 26, 2006

Abstract

A major discipline in software development these days is quality assurance. Quality assurance is a vast discipline, existing of many different areas of expertise. Software testing is a very important part of it. A variety of testing approaches and methodologies have been presented over the years, every one of them with proponents and critics. This thesis is about the selection and implementation of a suitable testing approach for the development of software components at Intersoft Software Research. Several existing testing methodologies and approaches are examined. These methodologies are compared to the development process used at Intersoft Software Research, and the most suitable testing approach is selected. The testing approach is adapted for the use at Intersoft Software Research and a tool environment is created to support the testing approach. The testing approach and tools have been used in the development of several components at Intersoft Software Research and the results of their use are analyzed in this thesis.

Table of Contents

List of Figures

Chapter 1

Introduction

In 2002 Intersoft Software Research (ISR) was established as a division of Intersoft to improve and support the software development of the business-class applications developed by Intersoft. Intersoft, located in Amsterdam, was set up in 1991 and started the production of an enterprise administration system, targeted at small to medium sized companies. The system evolved over time in response to client and market requirements and became harder and harder to manage due to unstructured growth of the system. In 2002, the development team realized that something needed to be done and they established a new division in Eindhoven: Intersoft Software Research.

Intersoft Software Research has strong links with the Eindhoven University of Technology. The development team at Intersoft Software Research mainly consists of graduates and students from the TU/e, the division was actually started by two graduates. The division's main goal is to improve the development of the business-class applications, by providing a component and library backbone for Intersoft's development teams in Amsterdam, and by assisting in the improvement of the development process in Amsterdam.

To support the development process of its components, Intersoft Software Research applies software engineering methodology and software development tools. This methodology is extensively described in a standards document and applied consistently. However, the methodology lacked a detailed testing approach which was recognized as a problem. Besides document and occasionally code reviews, there were no uniform validation and verification activities. Testing the code was only done at the developers own initiative, and not uniformly for all projects.

This thesis describes the introduction of a testing methodology supported by a tool environment. Specifically, the existing software development methodology used at Intersoft Software Research is extended with a testing approach, which is the first area of research. This testing approach should be focused on the development of components, since this is Intersoft Software Research's main area of development. The selected testing approach will be supported by a tool environment, which makes testing tools the second area of research. It is important that the testing approach and supportive tools strengthen each other to get the best results. A testing approach can be good in theory, but if there are no tools to support it, it might remain unused. Therefore, the selection and development of the testing tools is the most important part of the practical work.

1.1 Overview

Chapter 2 introduces some well known software testing methodologies and approaches. Four approaches are discussed that differ in intensity and strictness, some are more heavyweight and traditional, others are lightweight and more agile. Chapter 3 discusses the role of tools in the test development process. First, the tool categories that are important in testing are enumerated, then the advantages of both commercial and open source tools from these categories are discussed. The chapter is concluded with the results of a survey on the use of testing tools in business environments. Chapter 4 introduces some basic aspects of software testing that are important, regardless of the exact testing methodology that is chosen. The concepts discussed in this chapter are relevant for all software engineers that are involved in software testing.

In the subsequent two chapters specific implementation issues and results are presented. Chapter 5 explains the software development methodology that is currently used and which testing approach is the most suitable for this development methodology. The tool categories that are needed to support the chosen testing approach are also discussed. Chapter 6 gives an overview of the specific tools that are selected and developed. Furthermore, a detailed explanation of the proposed development and testing process in practice is given. Finally the conclusions of the graduation research are presented in Chapter 7.

Chapter 2

An Introduction to Software Testing Methodologies

In this chapter, general aspects of software testing are described, emphasizing the role of testing in different types of software engineering methodologies and processes. The role of testing in both heavyweight (or traditional) software engineering and lightweight (or agile) software engineering is discussed in more detail. The important properties of testing in the methodologies, their history and development over time are highlighted. More general testing terminology and properties are discussed in Chapter 4. In Chapter 5, a software testing process will be derived based on the methodologies described in this chapter.

2.1 Introduction

Starting around 1990, a new style of writing about testing began to challenge what had come before. Traditionally, either heavyweight test methods were used, or no testing at all. The seminal work in this regard is widely considered to be Testing Computer Software [Cem Kaner, 1993]. Instead of assuming that testers have full access to source code and complete specifications, these writers, who included James Bach and Cem Kaner, argued that testers must learn to work under more agile conditions of uncertainty and constant change. Meanwhile, an opposing trend toward process 'maturity' also gained ground, in the form of the Capability Maturity Model. The agile testing movement (which includes but is not limited to forms of testing practiced on agile development projects) has popularity mainly in commercial circles, whereas the Capability Maturity Model Integration (CMMI) was embraced by government and military software providers. The next sections examine these traditional and 'new' methodologies in more detail.

2.2 V-Model

The V-model (see Figure 2.1) is the traditional software engineering methodology and was originally developed from the waterfall software process model. The four development process phases – requirements, specification, architectural design and detailed design – have a corresponding verification and validation phase. The detailed design and code phase are tested by unit testing,

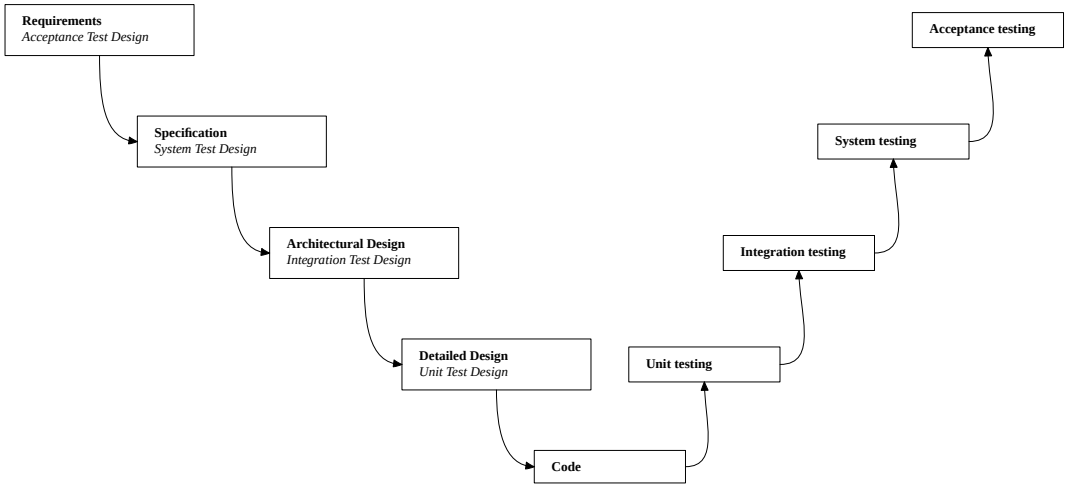

Figure 2.1: The traditional V-model, sequentially performing all test stages.

architectural design is tested by integration testing, system specifications are tested by system testing and finally acceptance testing verifies the requirements. The V-model gets its name from the timing of the phases. Starting from the requirements, the system is developed one phase at a time until the lowest phase, the actual code implementation phase, is finished. At this stage testing begins, starting from unit testing and moving up one test level at a time until the acceptance testing phase is completed. Moreover, the tests are not designed in the verification and validation phases but in the corresponding development phases. So a system test plan and system test design is already made in the specification phase.

Although the traditional V-model is easy to understand because of its sequential nature, this sequential nature also introduces some problems. How do you know how much effort to schedule for the integration, system and acceptance test phase? Once these phases are reached, how do you know how much longer they will take? How do you know that you have found all the defects that you can find? When is integration and testing really done?

Also because of this sequential timing of phases, where testing occurs after all design and implementation is done, it is not an adequate approach for the iterative software processes that are often used in today's rapidly changing environments. The V-model can be adapted to the iterative nature of some software development processes, which is also proposed in different ways in both [Marick, 2000] and [Jakobsson, 2003]. Both adapted approaches suggest to apply the validation and verifications phases from the V-model in parallel with the development phases and to perform the different test phases simultaneously in multiple iterations. This way the V-model testing method can still be useful in an iterative environment.

2.2.1 Spiral Model

Another iterative methodology that has evolved from the waterfall model and V-model is the spiral model which was first defined in detail by [Boehm, 1986]. As originally envisioned, the iterations are six months to two years long. Each cycle of the spiral starts with identifying the objectives and

risks of the portion of the product to be elaborated in that cycle. Each cycle includes the following steps:

- Determine objectives, alternatives, constraints.
- Evaluate alternatives, identify, resolve risks.
- Develop, verify next-level product.
- Plan next phases.

Many people consider the spiral model for big projects to be comparable to agile approaches for smaller projects. Most agile approaches tend to be more extreme than the spiral model. The most important advantage compared to the V-model is that the steps in the process are smaller and there is more space to adapt the development process after evaluating the feedback of earlier steps.

2.3 Capability Maturity Model Integration

CMMI [Software Engineering Institute, 2006], which is developed by the Carnegie Mellon Software Engineering Institute, is not really a software testing methodology but it is more a process improvement approach. It focuses on the standardization of processes and emphasizes the need to evaluate and develop these processes. Improvement of the development process will evidently lead to higher quality software. Version 1.1 of the CMMI for Systems Engineering and Software Engineering (CMMI-SE/SW 1.1) was released in December 2001. The CMMI replaced both the CMM version 1.1 of 1993 (SW-CMM) and the systems engineering standard EIA 731.

The model distinguishes approximately 25 specific process areas divided in 4 major areas: *Process Management, Project Management, Engineering and Support*. The CMMI provides two ways of measuring process improvement, namely the continuous and the staged approach. The staged representation uses maturity levels, which apply to an organization's overall maturity. Each maturity level focuses on a pre-defined set of process areas. The first level focuses on basic management practices and the improvement path for the organization is defined by the pre-defined sets of process areas for each successive maturity level. The continuous representation is one of the major improvements of the CMMI compared to the CMM. It uses capability levels, which define the quality of each specific process area. The improvements in capability levels are thus characterized relative to an individual process area. Targeting only those process areas that make sense in the individual context of an organization enables more flexibility and allows an organization to focus on risks specific to each process area.

Both maturity and capability levels are divided in five levels and are specified as follows:

Level 1 – Initial – Process unpredictable, poorly controlled and reactive.

Level 2 – Managed – Process characterized for **projects** and is often reactive.

Level 3 – Defined – Process characterized for the **organization** and is proactive.

Level 4 – Quantitatively managed – Process measured and controlled.

Level 5 – Optimizing – Focus on continuous process improvement.

The process areas that are most important and closely related to the traditional testing methodologies are *Verification, Validation, Process and Product Quality Assurance, Measurement and Analysis*. The best approach to focus on the improvement of an organization's test methodology using the CMMI model is to use the continuous representation and focus on the process areas mentioned above.

The CMMI is a heavyweight model, it requires a lot of investment for an organization to follow the CMMI standard. Before such an investment can be done, it must be guaranteed that the rewards are worth the investment. This is where the critics have their doubts. Gerold Keefer [Keefer, 2006] acknowledges that the CMMI standard is an improvement compared to its predecessors, but mentions a number of weaknesses of the standard. A very important disadvantage of the CMMI is that it is weak with regard to customer focus. For example, explicit customer feedback evaluation that plays a crucial role in other standards is hardly mentioned in the CMMI. Another deficiency of the CMMI is that due to its architecture it contains too much overlap between process areas. Redundancy makes standards difficult to understand, implement and maintain. Furthermore, there are simply too many process areas. A more hierarchical structure with process areas and sub-process areas could lead to a less fragmented situation. Keefer also states that one of the most important success factors of process improvement projects is 'senior management commitment', senior management must support the process improvement and believe that it is necessary. It is therefore a remarkable surprise that the stated responsibilities of senior management in the CMMI standard are extremely shallow. Christopher Koch's main concern, described in his article [Koch, 2004], is the integrity of companies that claim to have a certain CMMI level. Especially American and Eastern-European companies who want to outsource some of their development work, use CMMI levels to choose between companies. For these companies the stakes for a good CMMI assessment become higher all the time. It can be the difference between getting a contract or not. The danger is that companies use the CMMI for marketing purposes and not for process improvement. There are many stories of companies that exaggerate, or simply lie about, their assessment results. For example, claiming to have an enterprise-wide CMMI level, when only one project or department is assessed.

It is clear that the CMMI standard contains many good things and is able to improve the process of developing quality software, but there are also many drawbacks. Koch summarizes this as follows:

> "The depth and wisdom of the CMM itself is unquestioned by experts on software development. If companies truly adopt it and move up the ladder of levels, they will get better at serving their customers over time, according to anecdotal evidence. But a high CMM level is not a guarantee of quality or performance-only process. It means that the company has created processes for monitoring and managing software development that companies lower on the CMM scale do not have. But it does not necessarily mean those companies are using the processes well." [Koch, 2004]

2.4 Test-Driven Development

Test-Driven Development (TDD) [Beck, 2002] and [Astels, 2003] is an agile development method which encourages developers to develop code in small steps. Each step should add a small amount

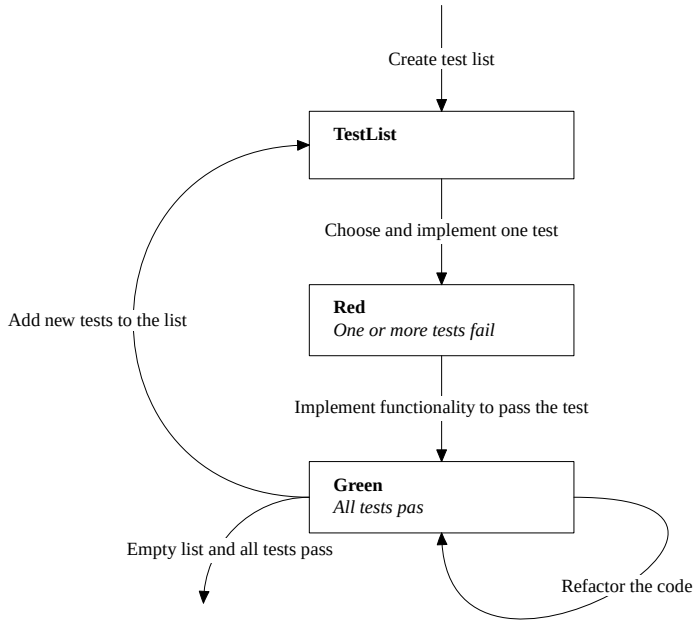

Figure 2.2: The steps taken in Test-Driven Development.

of new functionality. This is a well-known and accepted agile guideline [Ambpler, 2002]. Specific for TDD is the cycle in which this functionality is added, see figure 2.2. First a test is constructed, which should fail because the functionality is not added yet. Second, the simplest thing that could possibly make the test pass is implemented. Finally, the code is refactored to remove duplication or other code smells.

Refactoring is an integral part of several agile methodologies. Refactoring is a technique for improving the design of existing code, by applying a series of small transformations that change the structure and design of the code. Important to note is that refactoring should not change the external behavior of a program, nor fix any bugs or add new functionality. Refactoring should only be applied to improve the design and understandability of the code to make it easier to maintain and extend. A 'Code smell' is a hint that something might be wrong with the code. 'Bad Smells in Code' was an essay by Beck and Fowler, published as Chapter 3 of their book [Fowler *et al.*, 1999]. Many examples of code smell are discussed in this book, like duplication in code, methods or classes that are too big and long or vague method names. An interesting taxonomy for code smells was created by Mantyla [Mika Mantyla, 2005].

TDD is primarily a development method that has as a side effect that your source code is thoroughly unit tested. This is the same technique as that of Test-First Design. When a programmer has to write a test before the actual code, he needs to think of the interface that the code is going to use. So the actual design is specified by the tests that are written. An advantage of writing unit tests in general and especially writing unit tests using TDD, is that the tests extend the documentation. This is also indicated by Scott W. Ambler:

> "Like it or not, most programmers do not read the written documentation for a system, instead they prefer to work with the code. And there is nothing wrong with this. When

> trying to understand a class or operation most programmers will first look for sample code that already invokes it. Well-written unit tests do exactly this — they provide a working specification of your functional code — and as a result unit tests effectively become a significant portion of your technical documentation. The implication is that the expectations of the pro-documentation crowd need to reflect this reality." [Ambler, 2003]

An interesting side effect of TDD is that, theoretically, you achieve 100% statement coverage — every single line of code is tested — something that traditional testing does not guarantee (although it does recommend it). Of course, this only holds if the TDD technique is used very consistently. See Section 4.4 for more information on code coverage analysis.

2.5 Context-Driven Testing

The Context-Driven School (CDS) uses certain principles to approach software development. Basically, it states that it is not correct to assume that there is one single software development process that always works best. The best way to do things is dependent on the context of the project, which is not always the same.

The CDS applies the same principles on software testing as it does on software development. This is known as Context-Driven Testing (CDT), which was first introduced by Kaner, Bach and Pettichord [Cem Kaner, 2001]. The CDS summarizes CDT by 7 basic principles [School, 2001]:

1. The value of any practice depends on its context.
2. There are good practices in context, but there are no best practices.
3. People, working together, are the most important part of any project's context.
4. Projects unfold over time in ways that are often not predictable.
5. The product is a solution. If the problem isn't solved, the product doesn't work.
6. Good software testing is a challenging intellectual process.
7. Only through judgment and skill, exercised cooperatively throughout the entire project, are we able to do the right things at the right times to effectively test our products.

CDT is a software testing interpretation of the *No Silver Bullet* principle, which was introduced in [Brooks Jr., 1987]. They both state that there are no best practices that can be applied in all cases.

Thompson discusses an interesting nuance to CDT that tries to mediate between CDT and methodologies that propose 'best practices'. He explains his point of view as follows:

> "Several years ago, before Context-Driven became prominent, I tried to define 'Best Practice' for myself ... My answer turned out like a basic framework of things which I believe always apply to some degree, in some way. It consisted of a few basic principles, and elements within each principle. Some people say that is tantamount to 'Best Practice', but it doesn't quite resemble the traditional view. Then I read more about Context-Driven, and it occurred to me that my principles have much about context in them, yet claim to be absolutes. The word 'appropriate' features frequently. Perhaps

> if I were to call them ‘always-good’ practices, they might build a bridge between Context-Driven principle number two and the traditional Best Practice viewpoint.” [Thompson, 2003]

Thompson discusses a number of principles that he considers to be ‘always-good’, these are principles that must be considered in every project although they might not lead to exactly the same results in each case. Two examples of principles he regards as ‘always-good’ are ‘measuring test progress, possibly by coverage analysis’ and ‘use of appropriate techniques and tools to improve efficiency’. Thompson supports the CDT idea that some pragmatism in applying the ‘always-good’ practices is necessary, which may lead to different implementations of these practices in different contexts.

2.6 Conclusion

All the software development and software testing methodologies that are discussed in this chapter, have proponents as well as critics. All have been used in practice and seem to work out well in some cases. Each methodology contains some useful practices but, as stated by the context driven school, the best approach depends on the context. It is impossible to choose a methodology before knowing what software is developed, for whom it is developed, and who forms the project team. The CMMI and V-model, if used in the traditional way, are both heavyweight and imply much overhead in documentation for small project teams and companies. On the other hand, if the agile methods (CDT or TDD) are used, it is important to achieve a certain degree of formalism. In Chapter 5 the development process that is currently used by ISR is explained and the context is analyzed. A testing approach is proposed that is most suitable for the specific context.

Chapter 3

Tools in the Test Development Process

The previous chapter introduced a number of software testing methodologies and standards. Tools can be very helpful in applying such methodologies efficiently. This chapter discusses what different categories of tools are used in software testing. Especially the relation between the testing process and the tools is important in this regard. The difference between open-source and commercial testing tools is explained and a research study on the use of testing tools in commercial environments is elaborated. The selection of a satisfactory set of tools is described in Chapter 5 in parallel with the choice of the testing process or methodology to be used.

3.1 Introduction

Tools have been applied to support the process of documenting requirements, designing and writing code in Software Engineering for many years. The use of these tools has dramatically increased the productivity of the programmers. This has increased the pressure on testers, who are often the last ones to work on a product before the release and are, as such, perceived as the bottlenecks to the delivery of the product. As time to market has become more and more important in today's competitive environment, software testers are asked to test more and more in less time. Using tools is one way to improve the productivity and efficiency of testers.

3.2 Categories of Software Testing Tools

A tester does not use a single tool, but usually a complete set. There are various categories of tools that are useful in testing. Ideally each tester will use one tool out of every category, possibly ignoring the existence of some tool categories. Listed below are the most important tool categories. Note that not all tools are applicable to all fields of testing.

- **Functional / Correctness Testing Tools** – Often the most important tool in testing, because this provides the framework to implement and run test cases. It checks whether a component or method does what it is supposed to do and is often supported by a library of common test functions.

- **GUI Testing Tools** – Provides the possibility to playback specified user events on a GUI, using scripts or by recording them in advance.
- **Performance Testing Tools** – Measures, under controlled circumstances, the performance of a system at various loads. Generates input data, and measures performance in cpu-time, memory usage, etc. Very commonly used for testing web applications, database systems or other systems with multiple users or much input data. Also known as stress testing or load testing.
- **Test Management Tools** – Open frameworks providing features to allow collaborative testing like resource management, test planning, administration of test cases, logging results, etc.
- **Bug Tracking System** – Provides an interface for users and developers to report bugs and issues and for testers to give feedback. Supports multiple users, projects and components and often different versions and email notification. Acts like an electronic whiteboard.
- **Security Test Tools** – Provides often loosely coupled tools to find security weaknesses in networks, do password attacks, check for temporary files, or recognize code samples that are well-known potential security flaws.
- **Automated Testing Tools** – Automatically runs existing tests on a periodic basis to ensure that existing code is not broken by changes or newly developed code. Very useful in regression testing. Note that the tests are run automatically, not created automatically.

Of course, the list of testing tool categories is not complete, nor are all categories independent. There are many tools that cover more than one of the categories. It is obvious, for example, that a tool that tests for correctness also measures some basic performance statistics about the test run.

3.3 Commercial Tools

There are many commercial tool vendors that are completely dedicated to software testing and quality improvement. The big players in the software testing tools market develop complete software suites that cover the functionality of (nearly) all the testing tool categories described in Section 3.2. Some of the biggest players in software testing tools, or what they call ‘Quality Optimization Solutions’ and ‘Business Technology Optimization’, are, in random order, Segue Software, Mercury, IBM Rational Software, Compuware, and AutomatedQA. All of these vendors provide tools in most of the described categories.

Most of these complete packages are very extensive and expensive, many different testing and quality assurance activities are supported. The cost of acquiring the software itself is not the only thing that is expensive. There are also investments needed in terms of education and training of employees. Considering the complexity of these packages, it may be worthwhile to assign staff dedicated to testing only, instead of educating all software engineers in software testing. Besides training of employees, the development process has to be adjusted to work with the new tool, which requires investment of resources. After these changes, it might take some time before the development methodology can be applied efficiently again.

In general, acquiring a complete test package requires a big initial investment as well as an increase in resources needed to apply the development process. Such an investment may be worth-

while for big companies with big budgets, separate testing departments and the need for a broad collection of testing features. For smaller companies, who are searching for a software testing solution in a narrow context, commercial packages might be too expensive. Only a small subset of the features of such a package might be needed while the complete package is paid for. Often there are no resources to assign employees dedicated to software testing only. For smaller companies it is often more rewarding to search for dedicated testing tools that support a small subset of testing features.

3.4 Open-Source Tools

Section 3.3 concluded that commercial tools are often too extensive and expensive for medium or small-size companies. There is a need for dedicated tools that support a specific testing effort instead of a complete package. This is exactly the area in which the open-source community is active. Most open-source tools do not provide a complete testing suite but are specialized on a few tasks. Jim Rapoza gives a striking example of the problems with the cost of commercial testing tools in the field of web application development. Of course, the problem is not limited to web development only, but also occurs in other areas of software development.

> "A big problem with web application testing is that those who need the tools the most, such as independent site developers and departmental web site managers, can not afford the five-figure price for most of these testing tools. It's hard to tell the boss you need $20,000 to test an application you said you could build yourself." [Rapoza, 2003]

Several years ago, open-source testing software was usually limited to command-line tools that performed basic tests written in some scripting language. Since then, there has been vast improvement, to the point where many specific testing areas are supported by one or more open-source tools having a simple-to-use, capable GUI and meeting 90 percent or more of the needs of the testers. Obviously it can be very rewarding to have a look at open-source tools in the search for testing tools.

3.5 Survey on the Use of Tools in Commercial Environments

The selection of a proper set of testing tools is not easy. There are many alternatives and the selection is based on a number of criteria. To get an indication of the tools that are used in similar environments, a survey was conducted. This survey was published on the internet and posted on software testing forums and newsgroups, it was also sent to an academic mailing list. The survey contained a number of multiple choice and some open questions.

1. Which of the following development processes describes most accurately the process used by your company?
    - Waterfall approach
    - Iterative approach

- Extreme programming
- Formal methods
- Other

2. Does your company use tools to support functional or unit (non GUI) testing?

   - No testing
   - Manual testing
   - Custom made tool
   - Existing tool

3. Does your company use tools to support GUI testing? (See question 2 for multiple choice answers)
4. Does your company use tools to support memory (leak) testing? (See question 2 for multiple choice answers)
5. Does your company use tools to support regression testing? (See question 2 for multiple choice answers)
6. If your company uses existing tools, which are these?
7. Could you explain the choice to use existing tools or custom made tools? (What are the advantages and disadvantages of the tools that were chosen or discarded)
8. During a software project, how much of the total development time would be spent on testing (Don't consider debugging as testing)?

   - 0-10%
   - 10-20%
   - 20-30%
   - 30-40%
   - 40-50%
   - $>$ 50%

9. Some optional questions about the company and the function within the company.

A total of 60 completed forms were received. Most of the participants of the survey were visitors of one of the software testing forums [BetaSoft, 2006] and [Software Quality Engineering, 2006]. Because of this, the survey is not completely unbiased. Visitors of these sites are involved in software testing, or have a more than average interest in the subject. Therefore, these participants may not be representative for the average business environment. However, these participants probably know more about the subject of software testing than the average software engineer, which makes the results all the more interesting.

The intention of the survey was to assist in the choice of selecting a suitable tool set for the software testing process. In this regard, the partitioning of the development approaches of the first question on itself is not really interesting. The relation between the development approach and the answers on the other questions is more interesting. Is there, for example, a relation between the development approach and the way that testing is done? However, it is interesting to see that the iterative approach is the most used approach, see Table 3.1. This could make the survey more valuable, since this is also the approach used at ISR, see Section 5.2.

Waterfall approach	23%
Iterative approach	41%
Extreme programming	13%
Formal methods	9%
Other	14%

Table 3.1: Answers to question 1.

	No testing	Manual testing	Existing tool	Custom made tool
Unit testing (non-GUI)	5%	43%	42%	10%
GUI testing	3%	44%	53%	0%
Memory leak	29%	39%	30%	2%
Regression	2%	38%	55%	5

Table 3.2: Answers to question 2, 3, 4 and 5.

Notable is that there is a relation between the answers to question 1 and the answers to questions 2 up to and including 5. This can be seen in Table 3.3. People that use an iterative or extreme programming approach claim to use more tools and also spend more time on testing. This can be explained by the repetitive nature of these approaches. Testing needs to be done repeatedly which asks for a tool supported environment. However, the difference between the iterative approach and the other approaches is small. The tool support for regression testing in the iterative approach is surprisingly small. Regression testing is very important, especially in an iterative approach and is more than other types of testing suitable for automation and tool support. Some remarkable statistics:

- About 40% of the participants say they do regression testing manually. (Table 3.2)
- Unit testing is the only category worth mentioning where custom made tools are used. (Table 3.2)
- Memory leak testing is often not done at all, an explanation could be the use of programming languages with garbage collection. (Table 3.2)
- Most participants seem to do some kind of testing, but over 40% still does manual testing. (Table 3.2)
- An average of 35% of the total development time is spent on testing. (Table 3.4)

The use of existing tools was very diverse, the answers to question 6 did not show any surprising results. Most commercial products were mentioned a couple of times, but no tool stood out, nor was there a relation between the development approaches and the used tools. However,

	No or manual testing	Testing with tools
Waterfall approach	54%	46%
Iterative approach	39%	61%
Extreme programming	25%	75%
Formal methods	60%	40%
Other	50%	50%

Table 3.3: Relation between answers to question 1 and 2 till 5.

% of time on testing	0-10%	10-20%	20-30%	30-40%	40-50%	$>$ 50%
	4%	28%	17%	19%	22%	11%

Table 3.4: Answers to question 8.

the comments on why a certain tool was used (question 7) were interesting although not really surprising. Globally, there were two opinions.

1. Custom tools are very costly to develop and maintain, this is only advisable if the requirements are specific and you don't need a complete package with many different features.
2. Commercial tools are very expensive and it is hard to find commercial tools that satisfy all requirements. If there is money available, it is best to try to find (a combination of) tool(s) to satisfy all requirements. Furthermore, open-source tools are often available for specific tasks.

3.6 Conclusion

A number of tool categories that are useful in the testing process have been elaborated in this chapter. These are used in Chapter 5 to select tools from different categories that are needed to support the new testing approach. Furthermore, Sections 3.3 and 3.4 emphasize that commercial testing tools are very expensive. Open-source testing tools can be a good alternative, especially if there is no need for a broad package.

The survey presented in this chapter was conducted to get a better view of the use of testing tools in comparable environments and to assist in the choice of selecting appropriate tools. Although the survey might not be decisive in the selection of tools, it definitely resulted in some interesting statistics. It was surprising to see that, even with the participants coming mostly from software testing forums, there were still so many companies that spent little time on testing, or tested only in a primitive (manual) way.

Chapter 4

Technical Aspects of Testing

In this chapter some general aspects of testing are described. Chapter 2 already described testing methodologies and some properties of testing that are important in the methodologies. This chapter explains some terminology and technical aspects of testing that are important regardless of the testing methodology that is used.

4.1 Introduction

Software testing is an extensive field of quality assurance. The contexts in which software can be tested are various and within each context a possibly infinite number of test cases can be defined. To be able to narrow software testing to a more specific domain that is manageable and useful to ISR it is important to explain some basic aspects of testing and the way they are perceived in this document. First the ambiguous terminology of testing types is explained and the testing types are classified using three scopes. Then the matter of test case selection, namely how to narrow the infinite number of possible test cases to a manageable and useful set, is discussed. Finally, the value and limitations of code coverage analysis are described.

4.2 Testing Types

When reading on software testing in general, a great number of methods and techniques are often mentioned. In many cases there is a slight overlap or obscurity on the terminology that is used.

4.2.1 Taxonomy

In this document the following taxonomy is used to identify and distinguish different types of testing. This taxonomy of testing types is not complete nor does it cover the full vocabulary of testing terminology, but most testing types that are not in the list below can be placed in one or more of the described categories. The testing techniques are categorized using different contexts.

- Classified by purpose:

  - Correctness or functional testing
  - Performance testing
  - Reliability testing
  - Security testing

- Classified by life-cycle phase:

  - Requirements phase testing
  - Design phase testing
  - Program phase testing
  - Evaluating test results
  - Installation phase testing
  - Acceptance testing
  - Maintenance testing

- Classified by scope:

  - Unit or component testing
  - Integration testing
  - System testing
  - Acceptance testing

4.2.2 Important Testing Types at ISR

It is impossible to introduce all types of testing at once, nor is it necessary. This section explains which testing types are most important for ISR based on the three different classification contexts used in the taxonomy.

The most important testing type regarding the purpose of testing is *correctness* or *functional* testing, to test the correctness of the implementation. This kind of testing verifies whether the functional requirements are met. For some components, performance might also be important. In these cases it is important that the performance of the component can also be measured. Of course, the component must behave correctly if unusual inputs are used, which is a form of reliability testing.

Since ISR's main goal is the development of a component and library-backbone, the classification by scope can be narrowed to unit or component testing. Most components are developed fairly independent. If there are dependencies between the components, then these dependencies must be integration tested too. The integration of the component in the business application is not ISR's responsibility.

The testing types classified by life-cycle that are most interesting are program phase testing and evaluating test results. The requirements and design phase are already tested by reviewing the documents. The creation of the tests during the program phase and the evaluation of test results during and after this phase are very important. After that the life-cycle phases are less important, since the scope of testing is narrowed to unit or component testing. The last phase, the maintenance phase, is important again. In this phase bugs are reported and feature requests can be made. It is very important that bug fixes are tested and that the feature implementations are

tested in the same way as in the regular implementation phase. If bug fixes are implemented, it is also important to test that a bug fix does not introduce bugs in other parts of the component or application. Therefore, the tests developed in the regular implementation phase must be repeatable and usable as regression tests. Brooks describes this as follows:

> "Also as a consequence of the introduction of new bugs, program maintenance requires far more system testing per statement written than any other programming. Theoretically, after each fix one must run the entire batch of test cases previously run against the system, to ensure that it has not been damaged in an obscure way. In practice, such regression testing must indeed approximate this theoretical idea, and it is very costly." [Brooks Jr., 1995]

Currently most components developed at ISR are already tested by the developer. These tests are in most cases limited to functional testing in the programming phase at component level. In some cases performance tests are also developed, but this happens only occasionally and mostly for low level components. There are no guidelines to do so and this is completely dependent on the initiative of the developer. If the developer feels that the software is uncommonly slow, AutomatedQA's AQTime [AutomatedQA, 2004] is used to analyze the component for performance bottlenecks.

4.3 Test Case Selection

Section 4.2 describes different testing types and discusses which of the testing types will be most valuable for ISR. One of the most difficult aspects of software testing in general and in functional or correctness testing in particular, is to test the right things. There are many possible tests. In general a test has certain input values, performs some actions with these values and checks whether the output is as expected. In most cases the number of possible input values is very large, possibly infinite, and definitely too large to include a test method for each input. The choice of which inputs to use in a test method can make the difference between finding a bug or not. The next subsections contain techniques to select which input values to use in test methods.

4.3.1 Equivalence Partitioning

A test case selection technique that involves the identification of a small set of representative input values to invoke as many different input conditions as possible is *Equivalence Partitioning*. Combined with boundary value testing, see Section 4.3.2, this technique can be used to search for a limited set of input values which covers as many potential problem areas in functionality as possible. These techniques are black box testing techniques, since no detailed knowledge of the code is necessary to use this method of testing. Important input values to include in the test can be determined by examining the requirements.

All possible input values of a program are partitioned into equivalence classes. The partitioning is done such that a program behaves in similar ways to every input value belonging to the same equivalence class. To determine the equivalence classes, the input data and its requirements must be examined. Some examples of partitioning input values into equivalence classes:

- If the input data to the program is specified by a range of values, e.g. numbers between 1 to 5000, one valid ($1 \leq x \leq 5000$) and two invalid equivalence classes ($x < 1$ and $x > 5000$) are defined.
- If input is an enumerated set of values, e.g. $\{$a,b,c$\}$ one equivalence class for valid input values (a, b and c) and another equivalence class for invalid input values (all other values) should be defined.
- Partitioning of any input domain according to an If-Then-Else or Case Selection condition in the requirements (or in the code) results in an equivalence class for each part of the input domain, which are defined by the selection conditions.

4.3.2 Boundary Value Testing

After the partitioning in equivalence classes, a test should be made for one representative input value of each equivalence class. This value can be chosen randomly from each class. But just testing one random value is not enough, since typical errors in program functionality occur at boundary values which are not covered by a random value. This means that special testing is required at the boundaries of equivalence classes. Boundary values in general include values one before, one directly at and one after the boundaries of an equivalence class. This includes the minimum and maximum values of the input domain, since ($x < 1$ and $x > 5000$) can be represented by (MinInteger $\leq x < 1$ and $5000 < x \leq$ MaxInteger). In the example above, this would imply testing with input values $\{$MinInteger, 0, 1, 2, 4999, 5000, 5001, MaxInteger$\}$.

4.3.3 Multiple Input Parameters

Equivalence partitioning combined with boundary value testing as described in the sections 4.3.1 and 4.3.2 is a very straightforward and clear way to find useful test cases. But how does this technique apply to more complex situations? Consider a function that takes more than one input parameter. Each separate input parameter has its own set of possible inputs which can be partitioned into equivalence classes. After the partitioning in equivalence classes, the boundary values can be selected for each input parameter. As stated in the previous sections, a test case should be added for each input value.

Say there are N parameters with for each parameter M input values, then there are N to the power M possible combinations of input values in total. How many tests are necessary to test such a method adequately? The minimum number of tests in which we can use all test values is equal to M. Note that this concerns testing each value of each parameter once, and not testing all combinations of values of the different parameters. The problem with this approach is that it is impossible to say which parameter causes the failure if a test fails. It is better to keep all parameters constant and vary only the test value of one parameter at a time. This results in approximately $N * M$ test cases. This reasoning only applies if the *Single Fault Assumption* holds, which says: 'It is assumed that a fault, if it exists, will be exposed when a variable has a specific value, regardless of the value of the other variables.' This means that the parameters must be completely independent.

If the parameters are not independent, the *Single Fault Assumption* does not hold. The way that a function handles the value of one parameter is not always the same, depending on the other parameters. In this case it is not enough to use each input value a single time, but it is necessary to

test all combinations of input values for each parameter. The number of test cases needed to test such a function completely is M to the power N. Obviously, the number of test cases can be very large.

Applying equivalence partitioning and other test case selection techniques is even more important when the method to be tested has multiple input parameters, because the total number of possible tests is that large. It is up to the tester to analyze the specific case and decide which test cases to use. However, it is very important that the tester is aware of the fact that dependent parameters require more combinations of input values than independent parameters.

4.3.4 Error Guessing

In addition to the test cases and input values that are found with the black box testing techniques equivalence partitioning and boundary value methods, there are some other cases that should be tested. It takes some experience to define valuable tests using *Error Guessing*. This is a test case selection technique that can be considered as black box testing but is even more useful if the code can be inspected too (white box testing). *Error Guessing* means searching for input values that are likely to result in errors if the code does not handle them carefully, some examples are:

- Null values
- Extremely long values
- Almost correct values like, spaces in strings, quoted strings or all CAPS.
- Negative values
- Minimum and Maximum values.
- Values that should cause an exception (according to the requirements)

4.3.5 Test Case Selection at ISR

Currently there are no standards for test case selection at ISR. Developers who write their own tests do this on intuition and experience. In most cases this means that some of the techniques described above are used. Typically a developer tests different input values that, according to the requirements, should have different results, which is of course equivalence partitioning. And from experience a developer might test some uncommon values that are likely to result in unexpected behavior, which is Error Guessing. In the end, the test case selection is a result of the developers intuition and experience, and the willingness to critically and extensively test his / her own code.

4.4 Code Coverage Analysis

Part of the testing routine is the analysis of the degree to which a given test collection exercises a component's code, also known as code coverage analysis. There are different levels at which code coverage can be measured, the main ones being:

- **Statement Coverage** — Measures coverage of the lines of source code.

- **Condition or Branch Coverage** — Measures coverage of each condition or evaluation point.
- **Path Coverage** — Measures coverage of each execution path in a given part of the code.

All coverage measures have their strengths and weaknesses, which are extensively discussed in [Steve Cornett, 2005]. For general information on coverage analysis, see Patton's book on Software Testing [Patton, 2005]. Code coverage analysis is something you cannot do by hand. A tool is needed to do coverage analysis, especially if one of the more complex levels of coverage like branch coverage is measured.

4.4.1 Difficulties in Code Coverage Analysis

Measuring the coverage of tests can be very useful to show which parts of the code need more testing and to show possible vulnerabilities of the component that is tested. But there are some problems and weaknesses related to coverage analysis.

Black-Box Testing and Coverage

To test the full functionality of a component, it is important that the tests cover 100% of the component. But reaching 100% of coverage may be a problem if black-box testing techniques are used. Black-box testing means that the tests are created without a detailed knowledge of the component that is tested and tests are derived from the requirements and documentation. By testing the general functionality, derived from the documentation, a coverage of 80-90% may typically be reached. To complete the final 10-20% of coverage requires advanced knowledge of the components inner working. A 100% coverage goal is completely unrealistic if black-box testing techniques are used. Even with white-box testing techniques, reaching this goal may be very time consuming and might hardly result in finding any more mistakes in the code. It may be more worthwhile to perform a critical inspection on the remaining 10 or 20% of the code and to start error guessing, which may take a fraction of the effort to find the same amount of bugs. Furthermore, for non-critical code it may be more rewarding to spend the resources for different purposes and to risk leaving bugs in parts of the code that are hard to reach. After all, if a bug surfaces in the uncovered code after deployment of a component, then a regression test can be added while fixing the bug, which will cover parts of the code of the component that was not covered before.

Dead Code

Another thing that can prevent a test from reaching 100% statement coverage for any project is that a part of the code of a component may be unreachable, which is known as 'dead code'. This can be old code, that has become unreachable due to refactoring. Especially in TDD where refactoring is an explicit part of the development process, see Section 2.4, it occurs often that 'old code' becomes unreachable and it is forgotten to delete this 'dead' code. Code can also be unreachable because of dependencies between conditions in selection statements, or if a certain condition in an if-then-else statement always evaluates to the same value for another reason.

Proving whether code is unreachable is undecidable. In some trivial cases it is possible to prove that code is unreachable. For example, a method or procedure that is not called anywhere in the code is unreachable, many compilers even give a warning in such cases. It may also be possible to prove that a certain condition in a selection statements always evaluates to the same boolean value, resulting in unreachable code. But there is no algorithm that can prove, for any random program or random input, whether or not certain code is reachable. The results of statement coverage analysis may show that certain parts of the code are not reached, but from this it cannot be concluded that those parts are indeed unreachable. It is therefore a risk to delete code that seems to be 'dead'. On the other hand it is impossible to reach 100% if there is unreachable code. Keeping this in mind it is a good practice to write tests for all 'normal' functionality, which should cover a large percentage of the project's code and to critically inspect the remaining parts and examine whether they are reachable and whether extra tests are needed to cover them.

4.4.2 Test Quality

It is important to note that the coverage percentage of a test project is no measure for the quality of the test project or the component to be tested. A decent coverage is needed to test a component adequately, but it is no assurance that the component has been tested adequately. A test project can cover all the code of a component, but if only input values with a trivial outcome are used or if the results of the tests are not checked correctly then the tests are worthless and running them won't tell anything about the quality of the component that is tested. Therefore test cases should be constructed very carefully. It is not only the quantity of tests, but the quality of tests that matters. To create quality tests, it is very important to test with the right attitude. Especially when a developer tests his own code, it is tempting to test with a standard input to validate the correct working of the code. This is known as 'Test-to-pass'. Even if a developer tests his own code, it is important to stay critical and to test with input values that are more challenging for the program, see Section 4.3 on how to select these input values. This is known as 'Test-to-fail'.

Even if the quality of the tests is high, it is still not possible to conclude the absence of bugs. There is no point in time where you can say that there are no more bugs and the testing can stop. An interesting statement about testing and finding bugs at the end of the testing process was made by Beizer in his book Software Testing Techniques [Beizer, 1990]. He introduced the term *Pesticide Paradox* to describe the phenomenon that software becomes immune to tests if it is tested a lot. The term is used because of the similarity with insects that build up resistance against pesticide. Especially in iterative development models like the spiral model described in Section 2.2.1, this problem is likely to occur. The test process is repeated each iteration of the model. Each iteration, code is developed and the test are run. After a couple of iterations most bugs are revealed and continuing to run the same tests, or tests of the same kind, is unlikely to expose any new bugs. To find (some of) the remaining bugs, it is necessary to exercise new parts of the code and to test in different ways. This requires new, different, and possibly more complex tests.

4.4.3 Code Coverage at ISR

Currently code coverage analysis is not used in any way in the testing activities of ISR. This makes it very hard to evaluate the testing efforts of the developers. AutomatedQA's AQTime, which is one of the tools that is available at ISR already, does support code coverage analysis for Delphi.

So it is possible to introduce some form of code coverage analysis without much extra costs.

4.5 Conclusion

A number of general aspects of testing are discussed in this chapter. It is very useful for a software developer to be familiar with these aspects when he / she is to write tests of any kind. At ISR there were no standards on how to apply these principles in the testing activities at this time, although there were tools available to apply them. The quality of the tests that were developed completely depended on the personal skills of the software engineer and his / her experience with testing.

Writing high quality tests is not easy. There are many factors that determine the quality of the tests. To assist software engineers in their testing efforts and to give pointers on where to look for bugs, a Testing Standards Document (TSD) [Hermans, 2006] for ISR is written, which amongst others defines and explains the aspects of testing described in this chapter. The most important recommendations in the TSD are:

- Test to fail. Especially when a software engineer is testing his / her own code, it is important to have the right test attitude.
- Use code coverage analysis to analyze whether the quantity of tests is accurate. However, a good coverage does not imply good quality of tests.
- Use test case selection techniques to find a good set of input values, including less obvious ones that might result in unpredictable behavior of the program.
- Beware of the pesticide paradox. Try to test in different ways and use different paths through the code if possible.

Besides these aspects of testing, the TSD also describes the complete testing process that is proposed. In general all software engineers of ISR should read the TSD as well as the other standards documents, the Process Standards Document (PSD) and Coding Standards Document (CSD). Together they define the complete development process including the testing approach used at ISR.

Chapter 5

The Selected Testing Approach

This chapter describes the testing approach that is selected to extend the development process. First, the development process that is currently used is described in Section 5.2. The testing methodologies from Chapter 2 are compared to this development process to find the most suitable methodology. Section 5.4 describes the testing approach that is introduced and the kind of tools that are needed to support the testing. Chapter 6 explains how the proposed development and testing process is to be used in practice.

5.1 Introduction

Chapter 2 described some well known testing methodologies. The choice to adopt a certain test methodology depends on many things. The type of software product that is developed – are there lifes at risk if the software does not work correctly, or is it less critical? – and the size of the project and company are important properties. Also, the personal preference, affinity and experience of the employees with certain methodologies may influence the choice.

Regardless of which testing methodology is considered, it has to fit into the development process that is currently used at ISR. Of course this development process is not untouchable, it is constantly under review and evolves when better, or more suitable, practices are found. But the current development process is used for a reason and the developers will more easily adapt to a testing methodology that fits in the current practices and resembles the process that is already used.

5.2 The Current Software Development Process

ISR basically develops a component- and library backbone to support the development of business-class applications. The development of these components is supported by applying software engineering methodology and software development tools. The software engineering methodology that is used is described in a process standards document. Figure 5.1 shows the development model as it is depicted in the PSD [Zwartjes and van Geffen, 2004].

This development model is based on the European Space Agency (ESA) lite standard [Eu-

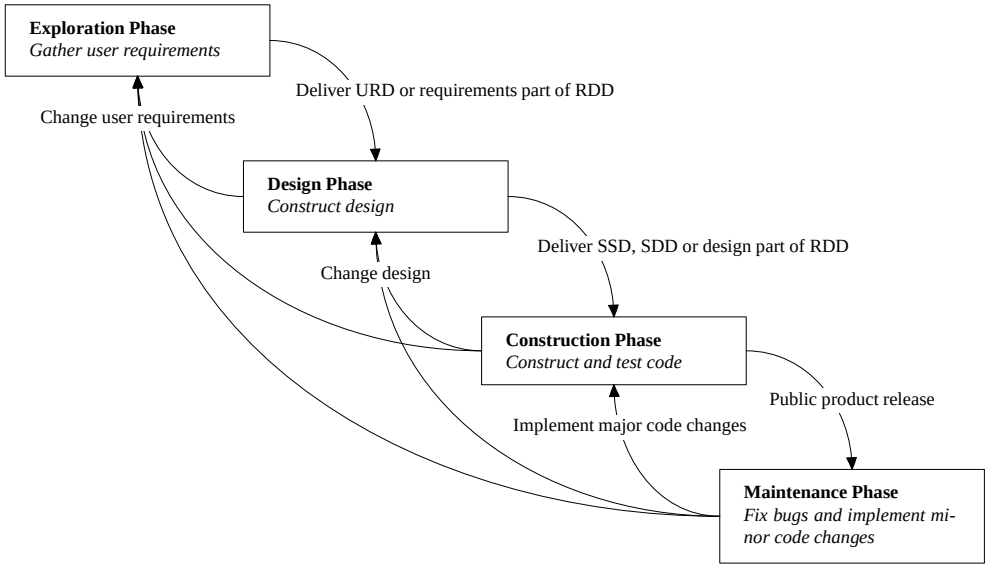

Figure 5.1: The model of the development process that is currently used at Intersoft Software Research, the relation between the phases in the process and the transitions between them.

ropean Space Agency, 1996]. Adaptability was the most important aspect that was incorporated into the standard. The following list from the PSD summarizes the key changes to the ESA lite process, to add adaptability [Zwartjes and van Geffen, 2004]:

- Management activities are essentially the same for each project, can be incorporated in the process, are minimized, and are documented in the process standard.
- No detailed upfront plan is made, the customer's wishes determine the schedule and can be adapted at any given moment.
- Adoption of an iterative approach and shortened release cycles. The possibility to fall back to a previous phase — without unnecessary restrictions and overhead — is added.
- Closer interaction with customer(s) and end-users; in each phase, customer(s) and end-users are highly involved.
- Less extensive product documentation, process focuses on actual product, not documentation.
- Single project team is responsible for the entire project, and all tasks are fulfilled by that single team.

The ESA lite methodology, in combination with these proposed changes, describes an agile development process. Preferably, the testing methodology should be agile too, or at least not impose more restrictions and overhead than the development process itself. Important features of the applied development process that influence the choice of the testing methodology are the iterative approach and shortened release cycles and the fact that a single project team is responsible for the entire project. The iterative development process implies the need for an iterative testing process.

A single project team being responsible for the entire project means that testing is preferably done by someone in the project team. Since the project teams generally consist of only one or two developers, it is obvious that each developer of a project also develops the tests for that project. There are no resources to add extra staff to each project group, so changing the development process is not an option in this case. The fact that the project teams are small already imposes that a testing methodology which is too rigorous and heavyweight should not be used.

5.3 Available Tools

Chapter 3 describes a list of tool categories that are useful in the software testing process. This section discusses the tools in these categories that are available, divided in commercial and open-source tools.

5.3.1 Commercial Tools

Two commercial tools have been purchased by ISR in the past. Taking into account the price of commercial tools and the limited resources to acquire new tools, it is very important to examine how the available tools can be of any use in the testing process. The two available tools are AutomatedQA's TestComplete and AQTime [AutomatedQA, 2004].

AQTime is a performance testing tool that has several profilers to do coverage, hitcount and timing analyzes and class profiling to find memory leaks. Currently the developers mainly use this tool to do memory leak testing and occasionally to find the bottleneck in components where performance is important.

TestComplete covers more software testing tool categories. It is a functional testing, GUI testing, automated testing and test management tool. Its main purpose is to build and automatically run test scripts for both GUI application and non-GUI components. This tool has not been used in practice yet. Besides some experimenting to explore its features, this tool can be considered to be 'shelf-ware'.

5.3.2 Open-Source Tools

Besides the commercial tools, there are open-source tools that can be used for free. An important advantage of open-source tools over commercial tools is that they can more easily be adapted or extended. This way a customized tool can be made that meets the requirements more accurately.

Currently, one open-source tool that matches one of the testing tool categories described in Chapter 3 is used at ISR. Namely phpBugTracker [SourceForge, 2003], which is an open-source web-based bug tracking system. This system is used frequently and seems to work properly. After deployment of a component the users, which are software engineers in most cases, are actively reporting feature requests, bugs and annoyances in the bugtracker and the developers responsible for the component usually respond in short terms to such reports.

Another interesting open-source tool in the functional or correctness testing category is JUnit [Erich gamma and Kent Beck, 2002] which is a unit testing framework. JUnit is developed by one of the pioneers of eXtreme programming and TDD, Kent Beck. It is a tool that supports

the creation and automated running of unit tests, which is indispensable when practicing TDD. JUnit is translated into and implemented in most popular programming languages nowadays, there are for example PHPUnit, CPPUnit, SUnit, and VbUnit for PHP, C++, Smalltalk, and Visual Basic respectively. There is also an implementation for the programming environment used at ISR, which is Delphi. Delphi's implementation of JUnit is DUnit [SourceForge, 2001]. DUnit is inspired on JUnit and was a rather straightforward port at first. Through the years DUnit has evolved and is improved to make better use of Delphi's specific constructs. Its further development was and is done by the SourceForge community [OSDN, 2004b]. The Borland Developer Network obviously appreciated the effort, because a version of DUnit was integrated in Delphi's Integrated Development Environment (IDE) starting from Delphi 2005.

When searching the internet for open-source tools, it is obvious that Borland Delphi is not widely used and supported by the open-source community. Collaborative development environments and online open-source software databases like SourceForge [OSDN, 2004b], Tigris [Collabnet, Inc., 2004] and Freshmeat [OSDN, 2004a] provide some Delphi tools and applications, but these are not as numerous as the tools for other platforms. Unfortunately this makes it harder to find open-source tools that are useful, or that can be customized to be useful, in ISR's development and testing process. The only open-source community dedicated to development in Delphi that is still active today seems to be the Jedi project [Jedi Community, 2004]. Actually, the JEDI Code Library (JCL) is used in the implementation of the practical work, see Section 6.3 for more information.

5.4 Introducing Testing in the Existing Development Process

This section describes how the existing development process is extended to include testing. First the important features of the current development process are summarized, then the testing methodology that fits best in this context is outlined. The specific testing activities that are used to extend the development process are derived from this testing methodology. Finally, the tools needed to support the testing activities are described.

5.4.1 Context of the Existing Development Process

The development process that is currently used is described in Section 5.2. The most important features of the process that influence the choice for the testing approach are the iterative approach with short release cycles, small project teams, relatively small projects and the fact that the process focuses on the product and not on the documentation. In Section 4.2 various types of testing were discussed. Section 4.2.2 concluded that the most important testing type for ISR is functional or correctness testing at unit or component level during the program phase.

5.4.2 Finding a Suitable Testing Methodology

From the methodologies described in Chapter 2, the CMMI and V-model approach are too heavyweight. Using these methodologies would require more project management and documentation than necessary for the relatively small projects at ISR. The spiral model described in Section 2.2.1

proposes an iterative approach, which is an advantage over the V-model, but the model is optimized for iterations from six months to two years and thus more suitable for the development of larger projects. The context driven approach states that the context of the projects must be analyzed to determine which test activities must be applied. In theory this approach makes sense, but it does not propose any testing process or testing activities in practice.

TDD is a development approach in which unit tests are developed in the programming / implementation phase, which is currently the testing type that needs the most attention at ISR. Furthermore, it proposes a process in which the software engineer develops the code and the tests for the code in an iterative approach. This means that the small project teams can develop their own tests without extra staff. Another advantage is that testing with TDD does not entail any additional documentation. The requirements and design of a component determine the tests to be implemented. No extensive test plan is needed because the next test that will be added is the test for the code that is going to be implemented in the current iteration.

One important property of TDD is that in each iteration the tests are written before the code. Some developers may like this technique, others may not. The actual order of coding and testing is not the most important thing. It is more important that the iterations are small. A developer should tackle only one problem in each iteration and should not proceed until a test is written to assure that the problem is solved by the implementation. This way, all the code that is implemented at a given point in time is always completely tested. The developer can proceed with confidence, and will be able to refactor the code and design without worrying whether the functionality is affected.

5.4.3 Tools Needed to Support the Introduced Testing Approach

It is very important to create a working environment that supports the TDD approach. The most important tool categories that are needed to create such an environment are a functional or correctness testing tool, to create tests, and an automated testing tool, to run the tests. Since TDD uses short iterations to write code and tests, it would be an advantage if the tests can be written in the same environment and programming language as the code. David Astels formulates this as follows:

> "There are a variety of frameworks available for writing unit/programmer tests in a variety of languages. The most effective are those that allow you to develop tests in the same language and same environment as the code being tested." [Astels, 2003]

Furthermore, it is important that the complete set of tests is executed repeatedly. It must be easy to run all tests of a component and see the test results immediately. Therefore, a tool that is integrated in, or at least compatible with, Delphi's IDE would be preferred. In his book, Astels suggests using the xUnit family of test frameworks. The Delphi variant of xUnit is DUnit, which is already discussed in Section 5.3.2. There were no resources to acquire a commercial testing framework, so the choice was restricted to: (1) adopting the open-source unit testing framework DUnit, or (2) develop a new customized testing framework. Both have their advantages and disadvantages.

1. **DUnit**

   \+ No development time needed.

+ The xUnit family is well known and has proved its value.
- Very extensive and undocumented code, so it is hard to adjust and maintain the tool for company specific needs.
- No manuals or help function available, so extra resources are needed to investigate how the tool works and to train the developers to use the tool.

2. **Develop a customized testing framework**

+ A well developed custom made tool can be easier to maintain.
+ A custom made tool can match the companies requirements better.
+ A manual can be made in parallel with developing the framework.
+ A customized framework may be profitable in the future, if it is sold.
- Requires a great deal of resources to develop a tool.

Eventually, the decision was made to develop a customized testing framework. The most important reason for this is the preference for a maintainable and adjustable tool, that matches the requirements better. This framework is called 'Project Autotest', and its development is discussed in detail in the next Chapter.

The tools that are currently used, AQTime and phpBugTracker, are also useful in the proposed testing approach. There will still be bugs, hopefully fewer than before, and they will need to be reported. The use of the performance profiler, AQTime, should not be limited to memory leak testing. It should also be used to measure code coverage; see Section 4.4 for more information on code coverage analysis. The profiler can measure code coverage at function, statement and condition level. In theory, if the TDD approach is used carefully, a constant condition coverage of 100% is maintained. In practice this may be difficult to reach, because AQTime measures condition coverage at assembly level of the code. Therefore, detailed knowledge of the programming language and compiler would be needed to achieve 100% condition coverage, but 100% statement coverage is realistic if the TDD approach is used. Not all components that are developed and deployed by ISR in the past are fully tested yet. The coverage profiler is particularly useful when a test is created for these existing components. The test progress can be measured by determining the degree in which a test project covers the associated component, and untested parts of the components can easily be found. Of course AQTime will still be used for memory leak and performance testing. Note that a high coverage percentage is also useful when a test project is used for memory leak testing.

5.5 Conclusion

The choice has been made to extend the agile development process, with an agile testing approach. There are a number of things that must be done to introduce the testing approach successfully. The approach must be explained to the developers, and the developers must be encouraged to actually apply it. To achieve this, the testing approach that is proposed is described in the TSD [Hermans, 2006], which is published at ISR and will be obligatory reading matter for all developers, next to the CSD [Zwartjes, 2002] and PSD [Zwartjes and van Geffen, 2004]. Furthermore, the actual implementation of the testing approach must be elaborated in more detail. A detailed description

of all practical work is discussed in Chapter 6, which also includes a scenario that describes the complete development and testing process in practice.

Chapter 6

Practical Work

This chapter discusses the details of the practical work. After explaining some testing methodologies, emphasizing important tool categories of testing and discussing general test principles, a testing approach has been proposed in Chapter 5. It also discussed the tools that are needed to support this testing approach and concludes with the decision to develop a custom made testing framework. This chapter will describe in detail the tools that have been developed.

6.1 Introduction

The testing approach that is proposed in Chapter 5 is an agile approach. It is based on TDD and uses an iterative approach to develop tests and write code in small steps. The tool that is needed to support this approach must allow creating and running the tests in the same environment as the code. The next section gives a detailed overview of the testing tool, which is called project Autotest and consists of several subcomponents. Sections 6.3, 6.4 and 6.5 elaborate the development process and implementation details of the various parts of the Autotest project. Finally, Section 6.6 describes a use case to show the proposed testing approach and the use of the tools in practice.

6.2 Project Autotest

Section 5.4.3 discussed the need for a framework to develop tests in the same language and environment as the code being tested. The type of tests that are going to be created with this framework are functional or correctness tests at component or unit level during the programming phase, see Section 4.2.2 for more information on testing types. Furthermore, it is important that the tests can be created easily, preferably using Delphi's IDE, and that it is easy to run tests quickly and often. A third requirement is that some of the tests must also function as regression tests. So it must be possible to store and re-run collections of tests, possibly automated. To achieve all this, project Autotest has been developed. Project Autotest has been developed from scratch but it is inspired on the xUnit family.

Project Autotest, see Figure 6.1, can be split into three parts. Component docAutotest, which is elaborated in Section 6.3, contains the actual framework that makes it possible to create tests in Delphi in a specific format and implements the basic functionality to run these tests. Application

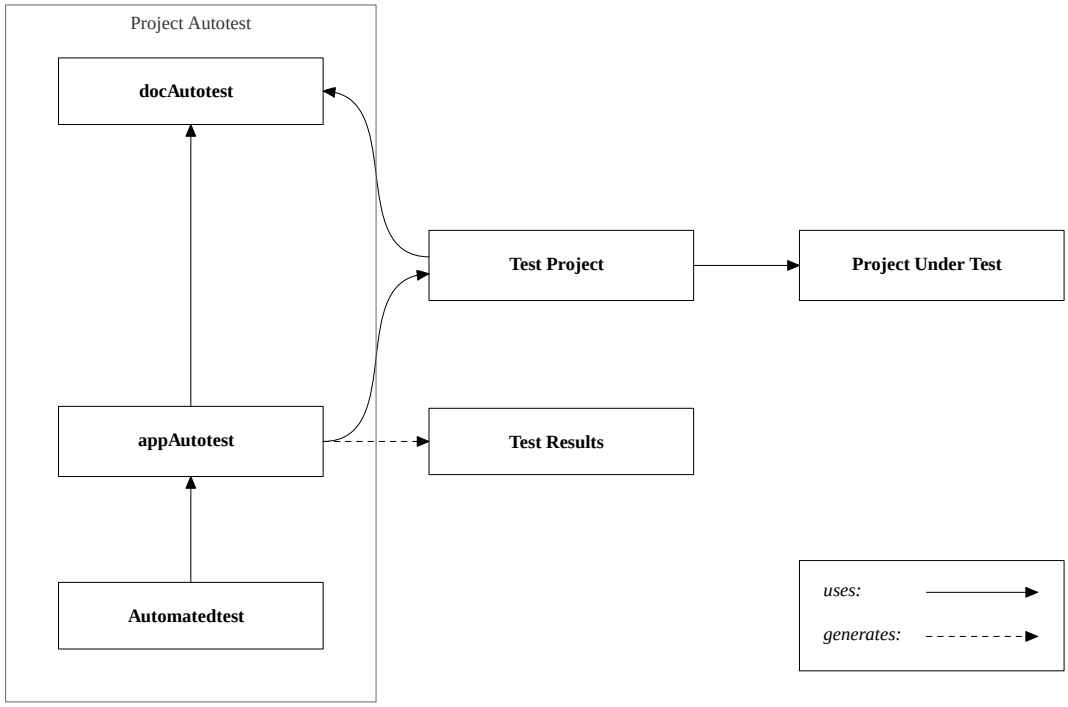

Figure 6.1: Overview of the custom-made unit testing framework, project Autotest.

appAutotest, which is discussed in more detail in Section 6.4, is the application that provides the interface to docAutotest to select and run tests and to collect the results of the tests. Automatedtest is the last part of project Autotest. Its main task is to enable the automated running of all regression tests made with project Autotest. Each part of project Autotest is elaborated in detail in the next sections.

6.3 docAutotest

Component docAutotest supports the development of tests. The component does this by providing a framework in which the test cases and test collections can be implemented and structured in a hierarchical way. This allows the developer to concentrate on creating the actual tests instead of the code that is needed to execute the tests. Furthermore, docAutotest has a library that contains a number of the most common and recurring functions that are used to create test cases. A manual for docAutotest is presented online on ISR's Wiki page, which is only accessible for employees of ISR. This manual explains several implementation details that are important when creating tests.

Figure 6.2 gives a logical overview of component docAutotest. The arrows indicate dependencies between subcomponents, dashed arrows indicate dependencies with external subcomponents. Subcomponent `Tests` contains the classes from which all test cases and test collections must be derived. This way all test cases and test collections conform to the same interface. Subcomponent `Runner` can execute all tests that comply to this interface. Tests are created as .bpl (Borland Package Library) files. These packages can be loaded dynamically to register all tests that are con-

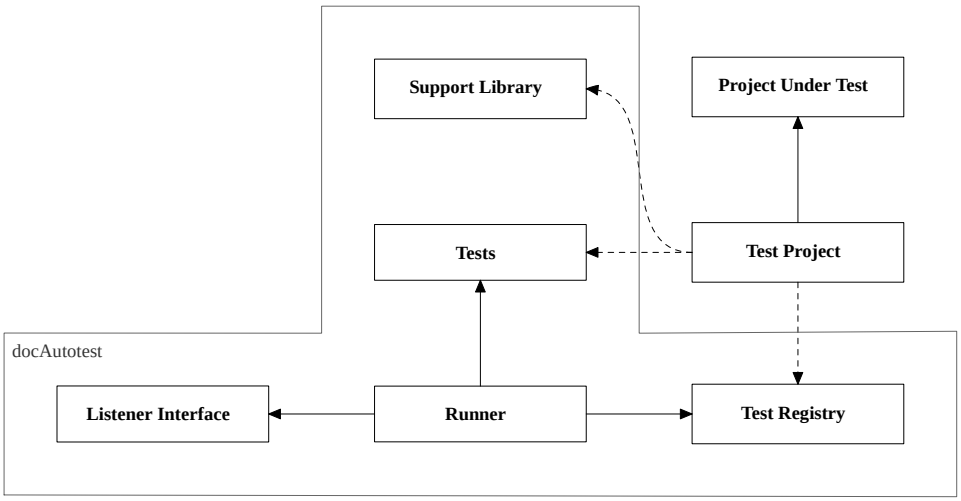

Figure 6.2: Overview of component docAutotest.

tained in a package with subcomponent `Test Registry`. This subcomponent is a global registry that is used by subcomponent `Runner` to find all the tests that are loaded. Subcomponent `Runner` iterates through all tests that are registered and creates messages for all different actions it performs. Examples of these messages are the start and end of a test case and the results of each test method. If a test method fails, it returns detailed information of the failure, see Section 6.3.4 for more information about test result messages. Subcomponent `Listener Interface` provides an interface that can be implemented to receive these test progress and result messages. Listeners that implement this interface can present themselves to subcomponent `Runner` to receive the messages. Component appAutotest, see Section 6.4, implements several of these Listeners. The last subcomponent is `Support Library`, this subcomponent contains common and recurring functions which can be used as building blocks to create test methods. These functions are compiled in a separate Delphi package, a .bpl file, to be able to reuse them in different test projects. The exact working of this subcomponent is explained in Section 6.3.3. The development process that is used to develop docAutotest is described in Section 6.3.1, the following sections contain a detailed explanation of each subcomponent of docAutotest.

6.3.1 Requirements and Design Development

Component docAutotest is developed according to the standard development process used at ISR. This process is described in the PSD [Zwartjes and van Geffen, 2004] of ISR and summarized in Section 5.2 of this document. The process is started with the user requirements phase followed by the design phase. In both phases a document is written, which are the User Requirements Document (URD) [Hermans, 2005c] and Software Design Document (SDD) [Hermans, 2005b] of docAutotest. During the implementation of component docAutotest, new requirements where recognized occasionally and eventually several iterations of adjusting the URD and SDD and changing or extending the implementation led to the final version of component docAutotest.

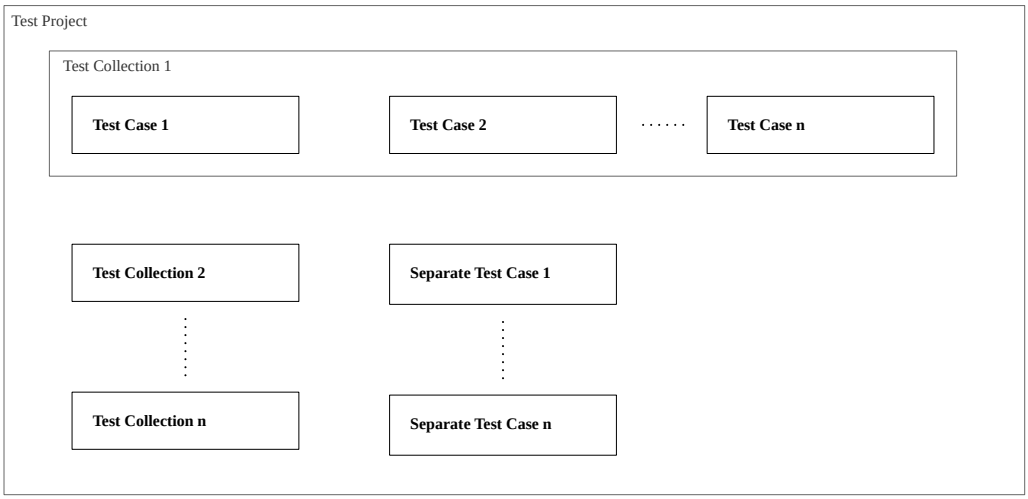

Figure 6.3: The hierarchy of tests in a test project.

6.3.2 Subcomponent Tests

Most approaches of testing impose a certain hierarchy on tests, defining suites, collections, cases etc. The test hierarchies that are used vary widely, also because the terminology of what a test case or test suite is, is not uniformly defined. `Tests` is the subcomponent of docAutotest that enforces a test project to have a certain structure, by implementing two base classes from which all test collections and test cases must be derived. The following terminology and hierarchy for the test projects at ISR is defined, see figure 6.3 for an overview of a test project.

- **Test Project** — In general the total set of tests to test a single project or library. As already stated, a test project is compiled as a Delphi package. This package consists of a set of one or more test collections and possibly some separate test cases. In Delphi, each unit or file has an initialization and finalization section that is executed if a package is loaded or unloaded. These sections are used to register the test collections and test cases with subcomponent `Test Registry`, so all tests are known to docAutotest by just loading the package that contains the tests.
- **Test Collection** — A test collection basically is a set structure which contains test case objects. This adds another level of structuring to the tests, only related test cases should be grouped in a test collection. Note that all test collections must inherit from the base class that is implemented in `Tests`.
- **Test Case** — A test case contains a number of related test methods that perform the actual tests. A test case class has a `Setup` and `Teardown` method, which are the initialization and finalization sections of the test case. All preparations for the actual tests are made in the `Setup`, test objects are created and initialized. `Teardown` does exactly the opposite, it cleans up after the tests and frees test objects. Note that all test cases must inherit from the base class that is implemented in `Tests`.
- **Test Method** — The elementary or atomic unit of testing. Each method performs a number

of actions and checks whether the required conditions hold after these actions. In most cases, this is done by comparing actual values of some variables or objects with expected values. The functions that are used to compare these values and to do other checks are defined in subcomponent Support Library, see Section 6.3.3.

Besides the hierarchy defined above, there is another way to distinguish tests. A test case can be used to do different kind of tests, therefore a type is assigned to each test case. There are four types, which are introduced for obvious reasons:

- **Development test** — Functional test written and executed while developing; should be used for test cases that are not yet ready to be included in an automated or regression test.
- **Regression test** — To test that changes in a project or one of its dependencies do not break the tested project. These tests are executed repeatedly and included in automated tests. Preferably, when a project is deployed, all test cases should be regression tests.
- **Performance test** — To test for speed or efficiency.
- **Experimental test** — To experiment and investigate while developing; typically used to try and experiment with different implementation options. These are temporary test cases which are not included in a release version of the project.

It is important that a test method is not too large and that each test method is independent. Test methods that are too big are dangerous because only the first check that fails is noticed and reported in test execution. After the failure of a check, the method is stopped and the program continues with the next test method. If there are ten checks in a test method and the first one fails, the last nine checks are not performed and the information that could be provided by these checks is lost.

Of course it is also important to keep the test methods independent. The actions of one test method should not change the variables that are checked in another one. Even sequentially changing a variable, performing some checks and restoring the variable afterward to its original value can influence other methods. When a variable is changed and one of the following checks fail the remaining part of the test method is not executed, so the variable is not restored to its original value. This can cause failure of other test methods if they assume that the variable has its original value. Dependent tests have to be performed in separate test cases, such that the tests always use the variables with their original values which are initialized in the `Setup`.

6.3.3 Subcomponent Support Library

The test methods that are discussed in the previous section are built from a sequence of actions and check methods. These check methods are defined in subcomponent `Support Library`. All check methods are defined in a similar way. There is one basic check method that is defined in the docAutotest package:

```
Check(Condition: Boolean; Msg: String);
```

This method evaluates the condition and raises an exception with the specified string message if the condition evaluates to False. Component docAutotest distinguishes between exceptions

raised by a check method and exceptions of any other type. This enables docAutotest to treat failures and errors differently. A failure occurs when a condition that is checked is different than expected, whereas an error occurs if a test, or the program that is tested, does not execute properly and raises any other type of exception. Both failures and errors are caught by subcomponent `Runner` which processes the results, see Section 6.3.4 for more information about this.

It is possible to define new check methods if they are convenient for new test projects. These new check methods must be implemented using the basic `Check` method defined above. To give an example, here is a method that checks whether an integer variable has the expected value:

```
class procedure TdocDefaultCL.CheckEquals(Expected, Actual: Integer;
  Msg: String);

begin
  Check(Expected = Actual,
    Format('Exp: %d, Act: %d. ' + Msg, [Expected, Actual]));
end;
```

This method is implemented by calling the basic `Check` method with the Condition parameter 'Actual = Expected'. Furthermore, the Msg parameter is concatenated with the formatted values of the actual and expected integer variables, to give a better indication of why the test failed. It is important to implement all variants of check methods by calling the basic Check method, this assures that each check method gives the same type of exception and enables subcomponent `Runner` to handle the failures and errors correctly.

Only the basic `Check` method is implemented in docAutotest. All other checks must be implemented in a separate package. These packages can contain check methods, but also other methods that are convenient to reuse. Such a convenience or utility method can be, for example, a method that initializes a large data structure, or a method that does a line count on text files. Both might be useful in different test projects. Each test project can define its own support library, but support libraries that are defined for other test projects can also be used. An extensive support library is implemented for docAutotest. This library will typically be used by all test projects. An overview of the check methods that are provided by this support library can be found below.

```
CheckTrue(Condition: Boolean; Msg: String);
CheckFalse(Condition: Boolean; Msg: String);
CheckEquals(Actual, Expected: NativeType; Msg: String);
CheckNotEquals(Actual, Expected: NativeType; Msg: String);
CheckInherits(Child, Parent: TClass; Msg: String);
CheckIs(TestObject: TObject; TestClass: Class; Msg: String);
CheckAssigned(TestObject: TObject; Msg: String);
CheckUnassigned(TestObject: TObject; Msg: String);
CheckException(Method: Procedure; Exception: ExceptClass; Msg: String);
CheckExceptionMsg(Method: Procedure; ExceptMsg: String; Msg: String);
```

The name and signature of these check methods implicitly explain what the checks do, which is one of the most important reasons to introduce them. The use of the correct check method

implicitly clarifies the intention of the test, which makes it easier to understand. It is also possible to make the failure messages more specific. The `CheckEquals` method, for example, formats the actual value and the expected value and automatically includes them in the exception message that is used if the check fails. The `CheckEquals` and `CheckNotEquals` methods are implemented for a number of native Delphi types, such as `Boolean`, `Integer`, `Extended`, `String` and `TClass`.

The `CheckException` and `CheckExceptionMsg` methods are somewhat different from the other check methods. These checks are defined to see whether the project that is tested raises the correct exception type or exception message in exceptional or unexpected situations. This can be done by implementing a procedure that gets the project under test in this exceptional situation and passing the procedure as a parameter to the CheckException or CheckExceptionMsg methods. These check methods execute the procedure and catch the exception that is raised by that procedure (if any), to compare it with the expected exception type or message. The following example shows the working of CheckException:

```
procedure RaiseDivideByZeroException;
begin
  x := y / 0;
end;

procedure TestDivideByZeroException;
begin
  CheckException(RaiseDivideByZeroException, EDivByZero, 'Assigning
    y/0 to x should raise an exception of the type EDivByZero');
end;
```

In the first procedure the exceptional situation is created, a division by zero. Actually, this piece of code won't even compile because the compiler already recognizes the division by zero. But it clarifies the example. The second procedure is the actual test, which checks whether the first procedure raises an exception of the correct type and otherwise reports a failure with the specified message.

Next to these check methods, the support library of docAutotest also contains some utility functions. At ISR, all workstations have Cygwin installed, which is a Unix-like environment for Windows. Cygwin contains a number of powerful commands and utilities. The support library of docAutotest provides a number of methods to enable executing cygwin commands and processing their results. With this Cygwin support, a number of utility functions are implemented, such as:

```
function GetLineCount(Filename: String): Integer;
function MatchesRegExp(s, RegExp: String): Boolean;
function Diff(Filename1, Filename2: String): TStringList;
```

These utility functions provide the opportunity to implement more complex check functions to check, for example, whether the number of lines in a file is as expected, and whether a string matches a regular expression:

```
CheckLineCount(Filename: String; Count: Integer; Msg: String);
CheckRegExpMatch(s, RegExp: String; Msg: String);
```

6.3.4 Subcomponent Runner and the Listener Interface

Part of docAutotest is also the `Runner` subcomponent that actually executes the tests. `Runner` retrieves all tests from the global registry and iterates through the test collections and test cases. For each test case sequentially `Setup`, all its test methods and `Teardown` are executed. Subcomponent `Runner`, nor any other part of docAutotest, displays any results of the tests. `Runner` creates a information message for every event that is of interest and sends it to all test listeners that have registered itself with the runner. These test listeners have to implement the interface that is defined in subcomponent `Listener Interface`. The events that are reported to the listeners are:

- The *Start* and *End* of a complete test-run.
- The *Start* and *End* of a test collection.
- The *Start* and *End* of a test case.
- A *successful* execution of a test method.
- A *failure* of a test method.
- A *error*, if one occurred in a test method.
- General *log-messages* that are specified in a test.

The messages that are sent to the listeners are straightforward text messages, except for the results of the test methods. The start and end messages of a test collection or test case only contain their class-name to identify the separate test collections and test cases, but the result messages contain more information. They include the filename of the file that contains the test method, the name of the test method, and the line number that indicates the beginning of the test method. This way each test method can easily be traced. Of course, in case of a failure or error, the exception message is also included in the message that is sent to the listeners. Delphi does not provide standard ways for tracing methods. Therefore, the JCL (see Section 5.3.2) is used to find the filename, procedure name and line number of the test methods. Implementations of the `Listener Interface` can be found in appAutotest, Section 6.4.

6.4 appAutotest

As described in section 6.2, appAutotest is the application that provides an interface to docAutotest to select which tests to run and to collect the results of the tests. Test progress and result information is gathered by test listeners, that are implemented conform the interface specified by subcomponent `Listener Interface` of docAutotest. The application provides three test listeners. There is a `GUI`, a `Console` and an `XML` test listener. The `XML` test listener is passive and only processes the information that is received from docAutotest's `Runner`. The `GUI` and `Console` listeners are interactive components, they are also test runners and provide an interface to customize a test run. Section 6.4.1 describes the development process that is used to develop appAutotest, which is slightly different from the one used for docAutotest described in Section 6.3.1. The subsequent sections describe the three implementations of the test listeners.

6.4.1 Requirements and Design Development

Application appAutotest is developed according to the standard development process of ISR, just like component docAutotest. Again, the process is started with the user requirements phase followed by the design phase. In the requirements phase no problems were encountered, and a URD was written. But in the design phase it became clear that the design was quite straightforward. The main reason for this is that the SDD's written by ISR do not contain the graphical design of the user interface. Only the architecture of the functional layer below is described. Often this is limited to describing a well-known design pattern and discussing the variations that are applied on the pattern for the specific case. This results in a SDD that contains a general introduction to the application, which is almost the same as the introduction section of the URD, and only a small section containing design specific information. Note that more extensive design documentation is generated with appDelphidoc, see [Zwartjes, 2003] and [Zwartjes, 2004], which generates Application Programming Interface (API) documentation from the code of a component. This documentation is available for users but not included in the SDD.

The duplication of general information in the URD and SDD was recognized as a problem by ISR earlier. Creating an SDD often started with just copy-pasting the introduction chapter of the URD. Updating one of the documents in the following iterations of the process, often led to inconsistencies in the introduction and definitions, acronyms or abbreviations. The need for a separate design document became more questionable with the actual design documentation being very little. As a result of this, a new document structure was proposed in the PSD that combined the requirements and the design into one document. This new document, the Requirements and Design Document (RDD), merges the introduction section of the URD and SDD and contains two separate sections for the requirements specific and design specific information. These changes in the process were applied during the development of appAutotest, which resulted in a RDD instead of two separate documents.

6.4.2 XML Test Listener

The `XML` test listener is a passive component. When the test listener is registered with docAutotest's runner, it receives notifications of all test progress and test result events. The `XML` test listener just processes all information and creates an XML document from it. The `XML` test listener is especially useful if tests are run automatically. XML can be used to publish the test results in a web-based system. The other two test listeners `Console` and `GUI` store the results only temporarily. When the application is closed, the results are lost. The `XML` test listener is the only listener that provides permanent storage of the results. The exact application of these XML test output documents is discussed in the following sections.

6.4.3 Console Test Runner

Running tests in the command line is a requirement for executing tests in an automated environment. Therefore, the `Console` subcomponent of appAutotest is very important. The most important responsibility of the `Console` version of appAutotest is not displaying the test results, but selecting and specifying which tests to run, and to specify an alternative way to process the test results. Here is an overview of the options that can be used for the `Console` test runner:

```
Usage: appAutotest.exe [option] ... [file] ...
Options: --execute, -e    Execute console version (do not load GUI)
         --verbose        Show verbose output (including log messages)
         --xml, -x [file] Report the test results in XML, optionally
                          specifying a path to the output file
         --help, -h       Print this help and exit successfully
         --version, -[vV] Print version information
         --test-cases, -t Only execute test cases of special type,
                          list of test case types: DEV,EXP,PER,REG
         --force-run, -F  Also run disabled test cases
```

The most important options are `-e` for distinguishing between the `Console` and `GUI` test runner and `-x` to register the `XML` test listener and specify the filename of the XML output document. If this command is used, the results are summarized in the command line and stored in detail in an XML document. Furthermore, there is an option to execute only test cases of a special type. This is useful, for example, if only regression tests need to be executed, which is the case in an automated environment. The `Console` test runner is used in Automatedtest, which is discussed in Section 6.5.

6.4.4 GUI Test Runner

The GUI part of appAutotest is the view that will be used the most while developing a component and the tests. Logically, the GUI version of the application provides more features and options to configure a test run than the console version. The GUI contains a number of panels with information about the tests. One panel displays which test packages are currently loaded, a treeview containing the structure of the loaded test collections, test cases and test methods, is shown in another panel. This treeview is also used to configure the tests. A Test collection or test case can be enabled or disabled by checking or unchecking it in the treeview. Furthermore, there is a panel that shows a log with the test progress and test result messages that are received from subcomponent `Runner` of docAutotest.

Of course, when tests are run, the most important thing to display are test results. But the list of progress and result messages can be very large and it will take some time to scroll through all results and see whether all tests succeeded. To make the global result of a test run instantly visible, a result bar is added to the application that is red or green after a test run. Of course, the bar is green if the test is successful. The application also provides some options to load and save test configurations, to edit the treeview and to enable XML output. The result bar is a key principle of TDD. Kent Beck refers to this as the TDD mantra:

1. *Red*—Write a little test that doesn't work, and perhaps doesn't even compile at first.
2. *Green*—Make the test work quickly, committing whatever sins necessary in the process.
3. *Refactor*—Eliminate all of the duplication created in merely getting the test work.

Red/green/refactor—the TDD mantra [Beck, 2002].

To integrate the testing into the development process, the GUI application can easily be started from Delphi's IDE. It is possible to assign appAutotest as a host application for a package that contains tests. When the host application and its parameters are configured correctly, it is possible to start appAutotest and run all tests with only one key-press.

Another helpful tool in the development of a component is AQTime, see Section 5.3.1. It is possible to use appAutotest as a host application in AQTime in the same way as in Delphi's IDE. When appAutotest is used as a host application in AQTime, it can be used to do memory leak, code coverage, or performance analysis. Application appAutotest runs tests on the project under test, while AQTime is profiling the test run. If the GUI test runner is used, it is possible to configure the test runs in detail which makes it possible to do coverage profiling on different levels. The coverage of a complete component can be measured using function, statement, or condition coverage. But it is also possible to see which part of a component is covered by a single test case. AQTime can be added to the 'tools' menu in Delphi's IDE which makes it more easy to access.

6.5 Automatedtest

Since docAutotest supports creating regression tests, it is important that these can be re-executed periodically. The regression tests created with docAutotest do not require any user input which makes them very suitable for automated running. For this purpose Automatedtest is introduced.

Ideally, Automatedtest should run all test projects developed at ISR daily. For this purpose, preferably a separate and clean computer system should be used. Running all test projects on a daily basis ensures that developers get feedback on the correctness of all components. This is important because there are dependencies between components, and it is important to detect defects that were introduced by changes in other components too. Testing with a clean system ensures that the latest versions of all components are used. Automatedtest can also be used by the developers, to test the locally installed components and to test unfinished code that has not been committed yet.

Running Automatedtest on a clean system can be divided into two parts. (1) A clean copy of the project tree containing all the components developed by ISR must be checked out from the Concurrent Versions System (CVS). All components must be compiled before the actual testing can start. (2) The actual testing consists of running the regression tests of each component and collecting the results. The test results must be reported in a clear way, summarizing the results and only emphasizing unsuccessful tests. Step (1) is discussed in Section 6.5.1 and step (2) is discussed in Section 6.5.2.

6.5.1 Autobuild

An autobuild system is used at ISR to automatically configure and build components. This autobuild system consists of two scripts and a number of modules, all written in Perl:

- **Configure** – A script that reads a set of project configuration files and creates a set of Makefiles to build the project.
- **Release** – A script that reads the project configuration files and deploys the files that are constructed by building the project.

The configure script is locally used by all developers to build both documents and source code. The variables that are unique for a project are stored in the project's configuration files. A project has a top level configuration file identifying and describing the project, a configuration file for each document, and a configuration file for the source code. For the release of projects, a separate computer is installed. This computer is a clean system which is not used for development, but only for releasing projects.

The autobuild system is perfectly suitable to function as a basis for the Automatedtest system. It already provides functionality to configure and build all components from a clean CVS project tree. Each component consists of a top level code library, with optionally a number of sub-libraries. It was possible to retain the existing Autobuild structure, because the test projects are compiled as Delphi packages, see Section 6.3. Each test project is added as a sub-library of the component it tests. Functionality was added to Autobuild to distinguish between a test sub-library and other sub-libraries, which was necessary because of dependencies between test libraries and their components, and the order in which they were built. Autobuild now builds all components with their general sub-libraries first, after which the test libraries are built.

6.5.2 Automated Running of the Tests

After successfully executing the Autobuild script, all components including their tests sub-libraries are correctly configured and built. Now, the actual testing can start. For this purpose another Perl script, named Autotest, has been written. This script uses appAutotest to execute the tests sub-library of a project, if it has one. Optionally, an XML document with the test results is generated and moved to a special results subdirectory of appAutotest's installation directory. When this script is executed in the top level of the project tree, the tests of all components are executed.

In general this Autotest script can be used by every developer to test the components that are stored locally, but it can also be used to run all test projects on a separate system. As already stated, all tests should ideally be run every day. For this purpose, the Autotest script can be used on the complete project tree to generate XML documents with the test results of all components. These results can easily be published online and an overview of all results can be presented on a so called 'test-dashboard', using eXtensible Stylesheet Language (XSL). This gives all developers a clear overview of the status of all components. Preferably, the system should also send an email warning to the responsible developer if there are tests of a component that do not pass.

6.6 Use Case: Developing a Component Using the New Approach

In this chapter, so far, the practical work that has been done has been presented and the development of the tools to support the testing approach has been discussed. But from this information, it is not directly clear how the testing approach and the tools influence the development of a component in practice. Therefore, a use case of the development of a component is presented, which describes the process step by step. A model of the development process is depicted in Figure 6.4.

1. Exploration phase: gather requirements and create a URD.
2. Design phase: make a basic design and create a SDD, the URD and SDD may be combined in a RDD.

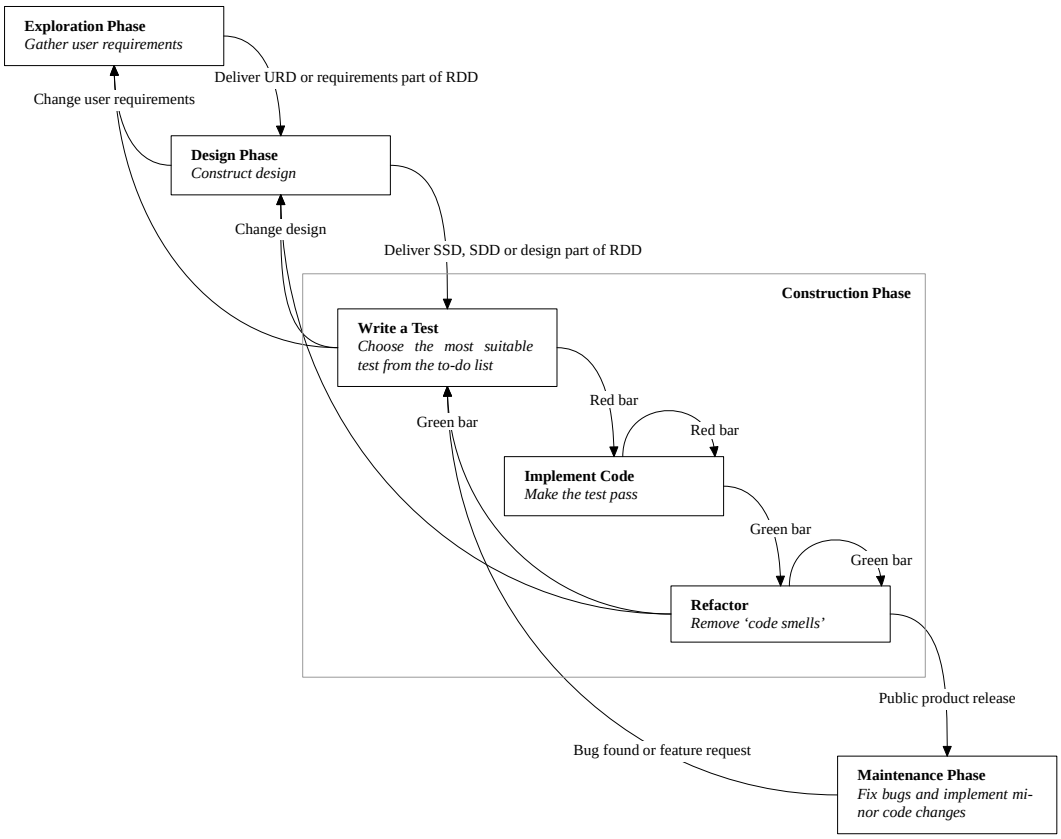

Figure 6.4: The model of the proposed development process including the testing approach, which is also described in the use case.

3. Construction phase:

   3.1 Create the directory structure for the new component in the project tree.
   3.2 Create the configuration files for the component and its test sub-library.
   3.3 Implement the basis of the design. It is difficult to start the TDD iterations with no code base at all. Therefore, it might be desirable to implement the declaration or interface of the basic classes of the design.
   3.4 Write tests for the current implementation, using docAutotest. Create a separate test support library for this component if necessary, containing specific check methods or other utility methods.
   3.5 Configure Delphi and AQTime.

   - Set appAutotest as the host application to run the tests sub-library.
   - Add AQTime to Delphi's tools and configure the parameters in such a way that it automatically runs the coverage analysis using appAutotest.

   3.6 Run AQTime to assure that the current implementation is completely covered, before continuing with the TDD phase.
   3.7 Create a to-do list with implementation tasks. Try to keep the tasks small. This list is initially derived from the user requirements and updated whenever a new task is thought of.
   3.8 Start of the TDD iteration:

   - Pick one task from the to-do list and write a test for it, using docAutotest. These tests are typed as development tests. Run the tests, the newly implemented test should fail: Red bar.
   - Implement the code to make the new test pass: Green bar.
   - Verify that the code is clean. If there are code smells, or you are not satisfied with the code as it is now: Refactor (run the tests again to make sure the refactoring did not break anything).
   - Occasionally re-run AQTime to make sure that the component is completely covered and nothing remains untested.

   3.9 When all requirements are implemented and the component is finished, run AQTime to check whether there are memory leaks and whether the performance is accurate. Refactor or improve the performance if necessary.
   3.10 If the component is deployed, all tests must be typed as regression tests, since only regression tests are run by Automatedtest.

4. Maintenance phase: re-run all tests regularly, preferably on a daily basis. Make sure that all regression tests pass if something is changed in the component or in any component that is related to it. When Automatedtest is used, the tests are automatically run each day.

   - When a bug is reported, write a test to reproduce the bug.
   - Write the code to make the test pass, which fixes the bug
   - Refactor again, if necessary.

   The same iteration can be applied when a feature request is done.

Of course, the development process is not completely sequential. Figure 6.4 also shows that it is possible to fall back to previous phases, so the requirements or design may be extended if necessary. This is especially important when a lot of refactoring is done, which may adjust the design. Note that the outgoing arrows from the construction phase only leave states with a green bar, so all tests should pass before changing phases.

This use case is based on the development of a new component. The testing framework is also used to develop tests for all existing components of ISR. In that case, the process changes a little. The first few steps can be omitted and the process basically starts with creating a to-do list, not with implementation tasks, but with test tasks. In each TDD iteration a small test is written which, if the current implementation is correct, should pass. Whenever a test fails, the implementation must be reviewed and corrected. Since not all code is tested yet, it is hard to tell whether changes in the code break anything. Therefore, refactoring should be done very carefully and not too rigorously. After each iteration, AQTime can be used to check whether a certain part of the code is covered accurately or whether more tests are needed. Eventually a coverage percentage of 80 or 90 % may be reached depending on the criticality of the component, see Section 4.4 for more information on coverage measures.

6.7 Conclusion

This chapter analyzes the practical work that was undertaken to develop the testing framework, project Autotest. In addition, the role of the testing framework and other tools in the development process in practice was presented in a use case.

In this documents a number of documents and tools are mentioned. A complete overview of all the tools and documents that are written and developed for ISR in this project are presented below:

- TSD [Hermans, 2006] — A document describing the selected testing approach and technical aspects of testing, see Section 4.5.
- docAutotest — The framework that enables writing tests, see Section 6.3.
  - URD of docAutotest [Hermans, 2005c]
  - SDD of docAutotest [Hermans, 2005b]
  - Component docAutotest, the code.
  - Test project docAutotestTests — The test project that tests component docAutotest.
  - User manual of docAutotest — A wiki system is used at ISR, which explains all the tools that are used in the development process. A page was added that explains the use docAutotest for the development of tests.
- appAutotest — the application that runs the tests and presents the results, see Section 6.4.
  - RDD of appAutotest [Hermans, 2005a]
  - Application appAutotest, the code.
  - User manual of appAutotest — A wiki page that explains the use of appAutotest in combination with AQTime.

- Script Autotest — The script that enables automated running of tests for multiple projects. Introduced to automate regression testing.
- Test project docDateTimeTests — The test project that tests component docDateTime, which provides functionality to perform date and time calculations.

So far, docAutotest has been used to write tests for a number of components. For most of these components, this work has consisted of rewriting existing tests using docAutotest. Currently there are two components that are completely tested, the tests cover all requirements of the components which resulted in a code coverage of approximately 90 %. Interesting to note is that in both cases the test packages are about half the size of the component measured in lines of code. Some components are partly tested, mostly because the developers only started using docAutotest halfway the development of the component and have not finished the implementation of the tests yet. At the very least, each component now has an empty test package, which compiles and can be run by appAutotest. This reduces the effort to start writing tests, for example if a bug needs to be fixed. Unfortunately, no component has yet been developed completely from scratch using the proposed development and testing approach, mainly due to a lack of time.

Overall, many positive reactions have been received from developers. Positive aspects of the new approach are reduced development times for tests, a uniform way of testing and improvement of general testing knowledge within ISR. Unfortunately, there were no resources available to accommodate a separate computer system which executes Automatedtest, see Section 6.5, on a daily basis. However, the script is ready to use and adds extra value to the complete testing process.

Chapter 7

Conclusion

Various testing methodologies have been examined and compared to the development methodology that is used at ISR. An agile approach was preferred over a heavyweight methodology, since the development process used at ISR is also agile. Therefore, the TDD approach was selected as the basis of the testing approach. Another advantage of using TDD in ISR's development methodology is that it only affects the construction phase, where the other phases basically stay the same. This makes it easier to adopt, since the developers do not have to change their whole development process. The proposed testing methodology is elaborated in a Testing Standards Document, which together with the already existing Process Standards Document describes the full development process. Besides the testing approach, the Testing Standards Document contains a section on technical aspects of testing, like test case selection and code coverage. Especially this last section with technical aspects of testing was received positively by the developers.

Supporting the proposed development and testing approach with an accurate tool-set was probably the most time-consuming and important part of the project. Making a testing approach easy to apply is absolutely necessary to make the approach successful and to get the developers to really apply it. Therefore, a testing framework has been developed that enables developers to create tests and code in the same development environment. The framework, project Autotest, has been applied retrospectively on six projects. The tool has been extensively used in the construction phase, which is the phase at which the proposed testing approach is aimed. Already, by simply translating the existing tests to tests based on the framework provided by project Autotest resulted in finding a number of previously undiscovered bugs. The reason for this is that it was easy to add some extra checks that were not done previously, the translation of the tests resulted in a more complete test package. Furthermore, the combination of project Autotest and AQtime, which enables developers to analyse the coverage of their tests, increases the value of the tool. Overall, project Autotest is considered as a very useful extension to the existing tool set. To integrate the new approach to testing more tightly in the development process, a script has been created to automatically execute all regression tests that have been created for all projects.

The main accomplishment of this project is that a gap in the already existing development process has been filled. Many methods have been presented to measure how good a development process is. An informal method is presented by Spolsky [Joel Spolsky, 2000], which boils down to evaluating whether a number of important aspects of software development are treated accurately at a company. Before the start of this project, the development process and tools used at ISR

already attended a number of these aspects, for example: a source control system was used, an autobuild system enabled building the complete component- and library backbone in one step, requirements and design specification were written and a bugtracker was used. At the end of the project, with the introduction of the testing approach, some important aspects that were unattended before can be added to this list:

- Functional tests are written in a structured and uniform way.
- No component will be deployed untested.
- An autotest system enables automated running of regression tests.
- Bugs are fixed before new code is written.

7.1 Future Work

The future use of the testing framework and whether its development will continue is somewhat uncertain due to financial problems at ISR. The head-office of Intersoft had to reorganize and economize due to disappointing sales of the end-product. As a result, it is still uncertain in what way the development of the component- and library backbone will be continued. However, the component- and library backbone is of significantly higher quality than the basis of the current end-product, which is still rather unstructured and chaotic. It might therefore be worthwhile to consider developing the end-product freshly again, using the component- and library backbone as a starting point. When this is done in a structured way, using the development process and testing approach used at ISR, this might result in a better product on the long run, than extending and debugging the current product. However, this might not be an option from a financial and business point of view.

Regarding the work on the selection of a testing methodology and the implementation of project Autotest, a few recommendations for future work are yielded. A very brief overview of some possibilities to improve and extend the testing process and tools is discussed here.

7.1.1 Automated Testing System

It is already possible to execute all tests, to export the results to XML and to present these results on a test dashboard online with XSL. So the tools and scripts necessary to accommodate a separate computer system that tests the complete component base of ISR on a daily basis are available, however such a system has not been composed yet. Having such a system would be a major improvement. It is very suitable to monitor the status of all the developed components and to give an overview of the work that has been done with project Autotest.

7.1.2 Test Support Libraries

Component docAutotest introduced test support libraries which contain check methods and other utility functions that can be used in tests. The support library that is provided by docAutotest mainly contains basic check methods. If more tests are created, the need might arise for other and more complex check methods and utility functions. Of course, creating tests will become easier when more and more comprehensive support libraries are available.

7.1.3 Code Generator

To create a test collection or test case, it is necessary to create a class with a number of standard procedures. These classes are basically the same for each test collection or test case except the class name. Therefore, these classes can easily be created with a code generator. Especially when it is implemented as an add-in in the IDE, it would be a major improvement in the usability of docAutotest.

List of Abbreviations

API Application Programming Interface
CMMI Capability Maturity Model Integration
CDS Context-Driven School
CDT Context-Driven Testing
CSD Coding Standards Document
CVS Concurrent Versions System
ESA European Space Agency
GUI Graphical User Interface
IDE Integrated Development Environment
ISR Intersoft Software Research
JCL JEDI Code Library
PSD Process Standards Document
PUT Project Under Test
RDD Requirements and Design Document
SDD Software Design Document
TDD Test-Driven Development
TSD Testing Standards Document
TU/e Technische Universiteit Eindhoven
URD User Requirements Document
XML eXtensible Markup Language
XSL eXtensible Stylesheet Language

Bibliography

[Ambler, 2003] Scott W. Ambler. *Agile Database Techniques*. John Wiley & Sons Inc, 2003.

[Ambpler, 2002] Scott W. Ambpler. *Agile Modelling, Effective Practices for EXtreme Programming and the Unified Process*. John Wiley & Sons Inc, 2002.

[Astels, 2003] David Astels. *Test-Driven Development, a Practical Guide*. Prentice Hall PTR, 2003.

[AutomatedQA, 2004] AutomatedQA. Performance profiling and memory debugging toolset. [online]. Available: `http://www.automatedqa.com/products`, 2004.

[Beck, 2002] Kent Beck. *Test-Driven Development, by Example*. Addison Wesley, 2002.

[Beizer, 1990] Boris Beizer. *Software Testing Techniques, Second Edition*. International Thomson Computer Press, 1990.

[BetaSoft, 2006] BetaSoft. Software testing and quality assurance online forums. [online]. Available: `http://www.qaforums.com`, 2006.

[Boehm, 1986] Berry W. Boehm. A spiral model of software development and enhancement. *ACM SIGSOFT Software Engineering Notes*, 11:14–24, August 1986.

[Brooks Jr., 1987] Frederick P. Brooks Jr. No silver bullet - essence and accidents of software engineering. *The Mythical Man-Month*, 4:10–19, April 1987.

[Brooks Jr., 1995] Frederick P. Brooks Jr. *The Mythical Man Month: Essays on Software Engineering*. Addison-Wesley, 1995.

[Cem Kaner, 1993] Hung Quoc Nguyen Cem Kaner, Jack Falk. *Testing Computer Software, Second Edition*. International Thomson Computer Press, 1993.

[Cem Kaner, 2001] Bret Pettichord Cem Kaner, James Bach. *Lessons Learned in Software Testing, A Context-Driven Approach*. Wiley, 2001.

[Collabnet, Inc., 2004] Collabnet, Inc. Tigris.org: Open source software engineering. [Online]. Available: `http://tigris.org/`, 2004.

[Erich gamma and Kent Beck, 2002] Erich gamma and Kent Beck. Junit. [online]. Available: `http://www.junit.org/`, 2002.

[European Space Agency, 1996] European Space Agency. *ESA BSSC(96)2 ISSUE 1: Guide to Applying the ESA Software Engineering Standards to Small Software Projects*. European Space Agency, 1996.

[Fowler *et al.*, 1999] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[Hermans, 2005a] John Hermans. *Requirements and Design Document: appAutotest*. Intersoft Software Research, 2005.

[Hermans, 2005b] John Hermans. *Software Design Document: docAutotest*. Intersoft Software Research, 2005.

[Hermans, 2005c] John Hermans. *User Requirements Document: docAutotest*. Intersoft Software Research, 2005.

[Hermans, 2006] John Hermans. *Testing Standards Document*. Intersoft Software Research, 2006.

[Jakobsson, 2003] Jakobsson. V-model testing – process model configuration using svg. April 2003.

[Jedi Community, 2004] Jedi Community. Project jedi. [Online]. Available: `http://www.delphi-jedi.org/`, 2004.

[Joel Spolsky, 2000] Joel Spolsky. The joel test: 12 steps to better code. [online]. Available: `http://www.joelonsoftware.com/articles/fog0000000043.html`, 2000.

[Keefer, 2006] Gerold Keefer. The cmmi considered harmful for quality improvement and supplier selection. *AVOCA GmbH*, Januari 2006.

[Koch, 2004] Christopher Koch. Bursting the cmm hype. *CIO Magazine*, March 2004.

[Marick, 2000] Brian Marick. New models for test development. *Quality Week '99*, March 2000.

[Mika Mantyla, 2005] Mika Mantyla. Bad code smells: A taxonomy. [online]. Available: `http://www.soberit.hut.fi/mmantyla/BadCodeSmellsTaxonomy.htm`, 2005.

[OSDN, 2004a] OSDN. Freshmeat.net. [Online]. Available: `http://www.freshmeat.net/`, 2004.

[OSDN, 2004b] OSDN. Sourceforge.net. [Online]. Available: `http://www.sourceforge.net/`, 2004.

[Patton, 2005] Ron Patton. *Software Testing Second Edition*. Sams publishing, 2005.

[Rapoza, 2003] Jim Rapoza. Open-source testers offer low-cost alternatives. [Online]. Available: `http://opensourcetesting.org/articles/2003_Aug11_Eweek.pdf`, August 2003.

[School, 2001] Context Driven School. The seven basic principles of the context-driven school. [Online]. Available: `http://www.context-driven-testing.com`, 2001.

[Software Engineering Institute, 2006] Software Engineering Institute. Capability maturity model integration. [online]. Available: http://www.sei.cmu.edu/cmmi, 2006.

[Software Quality Engineering, 2006] Software Quality Engineering. Interactive community exclusively engaged in improving software quality throughout the software development lifecycle. [online]. Available: http://www.stickyminds.com, 2006.

[SourceForge, 2001] SourceForge. Dunit: an xtreme testing framework for boland delphi programs. [online]. Available: http://dunit.sourceforge.net/, 2001.

[SourceForge, 2003] SourceForge. phpbugtracker: a web-based bug tracking system. [online]. Available: http://phpbt.sourceforge.net/, 2003.

[Steve Cornett, 2005] Steve Cornett. Code coverage analysis. [online]. Available: http://www.bullseye.com/coverage.html, 2005.

[Thompson, 2003] Neil Thompson. 'best practices' and 'context-driven': Building a bridge. *STAREast paper*, May 2003.

[Zwartjes and van Geffen, 2004] Gertjan Zwartjes and Joost van Geffen. *Process Standards Document*. Intersoft Software Research, 2004.

[Zwartjes, 2002] Gertjan Zwartjes. *Coding Standards Document*. Intersoft Software Research, 2002.

[Zwartjes, 2003] Gertjan Zwartjes. *User Requirements Document: appDelphiDoc*. Intersoft Software Research, 2003.

[Zwartjes, 2004] Gertjan Zwartjes. *Software Design Document: appDelpiDoc*. Intersoft Software Research, 2004.