Software Process Improvement: Continuous Integration and Testing for Web Application Development

Matias Muhonen

University of Tampere
Department of Computer Sciences
M.Sc. Thesis
Supervisors: Eleni Berki and Timo Poranen
3.6.2009
Abstract
Software testing is not the only approach to improve software quality but, perhaps, one of the most important ones. This is because testing gives us the confidence that the software will work as it should in its intended environment. In this thesis the author introduces a Software Process Improvement (SPI) technique for conducting automated web application security testing. The continuous integration process provides rapid and automatic feedback on the security of the web applications under development. Continuous integration has improved the software development process of a company specialized in web-based systems. We found out that the testing technique we suggested can be used as an additional testing mechanism with the existing approaches.

Keywords: Software Process Improvement (SPI), software quality, testing, continuous integration, web application security assessment.
ACKNOWLEDGMENTS

This thesis was not only an effort of my own but involved many people and organizations. First, I want to thank Henri Sora who proposed to initiate the project. Ambientia Oy provided funding for the Continuous Testing project. My colleague Mike Arvela greatly helped by programming many components used in the process.

Eleni Berki and Timo Poranen from the University of Tampere kindly supervised the thesis and provided guidance which I am very grateful for. I want to thank my parents and my fiancée Chienting for always supporting me.

Tampere, June 2009
Matias Muhonen
CONTENTS

1 Introduction 1

2 Background 4
   2.1 Software development processes ................. 4
      2.1.1 Royce’s model ......................... 5
      2.1.2 Prototyping .......................... 6
      2.1.3 Spiral model ......................... 8
   2.2 Software development methodologies .......... 9
      2.2.1 Agile software development .............. 10
   2.3 Different views of quality .................... 12
      2.3.1 Software quality ...................... 13
   2.4 Software testing .......................... 15
      2.4.1 V-model .............................. 16
      2.4.2 Testing approaches .................... 18
      2.4.3 Test automation ....................... 19
      2.4.4 Approaches for automating test case design ... 22
   2.5 Summary .................................. 23

3 Review of the literature 24
   3.1 Software integration .......................... 24
   3.2 Continuous integration ....................... 25
      3.2.1 Continuous testing ..................... 29
      3.2.2 Atlassian Bamboo ...................... 30
   3.3 Web application security testing ............ 31
      3.3.1 Web applications ...................... 31
      3.3.2 Web application security vulnerabilities .. 34
      3.3.3 Web application security assessment tools .... 37
      3.3.4 Acunetix Web Vulnerability Scanner ....... 40
   3.4 Summary .................................... 41

4 The proposed Software Process Improvement 42
   4.1 Background ............................... 42
      4.1.1 Motivation ............................. 42
      4.1.2 Business requirements .................. 43
      4.1.3 Company’s project model ............... 44
   4.2 Software process improvement ................. 46
4.2.1 SPI I: executing commit builds ................. 47
4.2.2 SPI II: executing functional builds ............. 49
4.2.3 SPI III: executing security scanning ............ 52
4.2.4 SPI IV: publishing vulnerability reports ........ 55
4.3 Summary ........................................... 58

5 Conclusions ........................................ 59
5.1 Project retrospect .................................. 59
5.2 Future work ....................................... 60

References ............................................. 61
1 Introduction

Background: The World Wide Web (the web) is a global system of hypertext documents. From the very beginning, most content on the web was static. That is, the same content was displayed in response to all page requests. The number of web users increased during the 1990s. Conventional desktop applications were begun to be developed as web applications. Web applications differ from static content so that the content can vary per page request basis. For example, a weather application could display weather information for a certain city according to the user’s choice.

The popularity of web applications introduces new challenges. Online banking users assume that handling financial matters online is as safe as visiting a banking center. Web applications must be reliable: for example, in electronic health services, patients’ privacy and data security must be the same as when visiting a clinic. Compared to this ideal situation, security vulnerabilities has been found from web applications. Vulnerabilities allow attackers to exploit the functionality of web applications. For example, a vulnerability was found from Sampo Bank’s online bank that allowed embedding external content [Digitoday, 2008]. The vulnerability allowed attackers to hijack user information with phishing attacks [Li et al., 2007].

The vulnerabilities discovered demand solutions for the problem. Vulnerabilities like other types of software errors are caused by programming errors. The goal of software development is to develop software, which has as few errors as possible. This goal can be targeted with many approaches. One of the approaches is software testing.

Testing is a process of executing a program with intent of finding an error [Myers, 1979]. An error is a behavior that differs from the expected behavior of a program. A typical error is an unexpected crash of a program. A program can be tested with a black or white box approach. In black box testing a program is tested by its designed functionality without paying attention to the internal structure and implementation of the product. In white box testing, the tester constructs the test cases with the knowledge of the internal structure of a program.

There are different types of web application vulnerabilities which must be addressed with security testing. The highest risks are currently caused by cross-site scripting (XSS) and SQL injection vulnerabilities [Lucca et al., 2004; Kals et al., 2006; CVE, 2009]. The latter allows embedding external content to a web application. SQL injection might allow an attacker for example to bypass
a user authentication. Security testing allows testers to uncover these and other types of vulnerabilities. In practice security testing should be performed when a program is being developed so that the vulnerabilities can be fixed before the web application is released and can be attacked.

Security testing can be performed manually by mimicking an attack or the testing can be automated with a security testing tool. These days, most security testing is aided with a security testing tool. A web application scanner is a tool which simulates an attack to a web application [Curphey et al., 2006]. The scanner forms specifically formatted page requests which test, for example, XSS vulnerabilities and SQL injections. If the scanner detects an unexpected response, a vulnerability is supposed to be found [Huang et al., 2003].

Rationale: In this thesis, the author developed the continuous integration process to reduce the risks related to various types of security vulnerabilities. What usually remains after individual software components are completed is a number of integration problems. Royce [1999, 12] calls this phenomenon "integration nightmare" due to unforeseen implementation issues and interface ambiguities. Integration nightmare is not only an engineering dilemma: the integration and test stage can account as far as 40% of expenditures. An approach to ease integration problems is to integrate more often. From the practice of short integration cycles results what is called continuous integration.

There is empirical evidence [Miller, 2008; Cannizzo et al., 2008] that continuous integration as a software engineering quality practice benefits software development projects. If integrating more often eases integration problems would the same principle apply to testing? We had the idea that continuous integration could be "extended" with automated security testing. This approach is not, indeed, new [Cannizzo et al., 2008]. Large scale automated testing with continuous integration seems to benefit software development. However, we were unable to discover attempts of such approach for web application security testing.

The continuous integration process provides rapid and automatic test feedback on the security of the web applications under development. The testing process includes black box scanning to uncover vulnerabilities. We show that this kind of process is suitable and implementable in an organization which prefers rapid, constant and automatic feedback. In general, the process we implemented might be beneficial in organizations favoring iterative development with frequent releases of software.

This thesis is structured in four chapters. First, the theoretical background of this thesis is briefly introduced in Chapter 2. The notions of software process and
software development methodology are discussed. Furthermore, we make notes on software quality. Also, software testing and test automation as an approach to build and check software quality are explained.

A literature review is conducted in Chapter 3. The topics related to this study are continuous integration, continuous testing and web application security vulnerabilities. In addition, the author explains how to construct SQL injections and cross-site scripting vulnerabilities. Furthermore, security testing tools are presented as an approach to uncover web application vulnerabilities.

In Chapter 4, we propose a software process improvement technique for continuous build testing. The business requirements concluded us to choose two tools for continuous integration and security testing: Atlassian Bamboo and Acunetix Web Vulnerability Scanner. Four software process improvements are introduced: (I) executing commit builds, (II) executing functional builds, (III) executing security scanning, and (IV) publishing vulnerability reports.

In Chapter 5, we evaluate the experiences of implementing the build testing process. Also, further remarks of future improvements for the process are made.
2 Background

2.1 Software development processes

At the very basic level, developing a computer program could be described with two activities: analysis and code. Regardless of the program to be developed, the programmer first needs to get familiar with the target application domain and then write the code to solve the problem (assuming the problem is solvable with programming). Royce [1970, 328] calls this two-step process of creating a computer program analysis and coding.

Boehm’s [1988, 62] finding is that the earliest software process model is the code-and-fix model. First, the programmer writes some code, and then, she fixes the problems in the code. However, the code-and-fix model does not state the fact that the programmer needs sufficient knowledge of the target application domain before she can begin programming.

Both analysis and coding and the code-and-fix models could be described as ad hoc development. Boehm [1988, 62] identifies two main difficulties for ad hoc development. Firstly, the code lacks higher level structure. Subsequently, it becomes expensive to further develop the program because of lacking structural design. Secondly, even well-designed software has had problems to match customer needs and expectations.

To overcome these difficulties, the concept of a software process was introduced. The primary function of a software process is to (1) define stages of software development activity, (2) establish the transition criteria from progressing from one stage to the next. These transition criterion should include conditions for completing the current stage as well as entrance criteria for the next stage. A software project model should answer the two fundamental questions: (1) What shall we do next? (2) How long shall we continue to do it? [Boehm, 1988, 62]

Clearly, when evaluating ad hoc development, it lacks well-defined stages. The transition criteria are also nonexistent because the stages are not well-defined. As it was evident that more formal development process models were needed, sequential development processes emerged. They attempt to model software development with a sequence of subsequent activities. These development activities should not overlap in any occasion, so for any sequential process it can be said that the process is distinct in one stage or comprises one activity (such as testing but not programming). In the following sections, we introduce a few of these sequential software development processes.
2.1.1 Royce’s model

As described in the previous section, software development can be seen as a sequence of subsequent activities. Royce’s [1970] model includes seven distinct software development activities ordered in linear fashion: system requirements, software requirements, analysis, program design, coding, testing and operations (Figure 2.1). Requirements elicitation was the first activity. With subsequent steps the design proceeds from abstract to more detailed level. Coding and testing were and have been the last stages of the process. Royce [1970, 329] considered risky that the first real (perhaps customer) validation of work is done as the last action.

![Figure 2.1 Implementation steps to develop a large computer program for delivery to a customer [Royce, 1970, 330]. Royce’s model bears similarities to the widely used waterfall model until today.]

Royce’s model implies that the software requirements should be clearly known before any actual implementation of the software product in question. Also, the fact is that the rest of the activities in Royce’s model depend on the requirements elicitation and its validity. This design-before-coding principle emphasizes that
problems should be solved in conceptual level before actually implementing the design. This may set high demands for customer knowledge and may cause difficulties in early phases of a project.

Still, as linear as Royce’s model seems to be, it is not the whole truth. Fallbacks may happen in two ways. (1) Progress falls “back” for more than one step. (2) Progress is reverted by to the preceding step. Royce considers the former not desirable [Royce, 1970, 329]. If the testing stage reveals invalid software requirements, the effect may not only be invalidation of some implemented parts but also analysis and program design may need to be redone. This implies cost and schedule overruns. However, if there is an iterative relationship between the succeeding stages (such as coding and testing), the change process is scoped down to manageable limits [Royce, 1970, 328].

2.1.2 Prototyping

What we notice from Royce’s software development process is the early emphasis on software requirements. We made an implication that the process depends on the requirements elicitation and its validity. This places a risk for a project if requirements are misunderstood. We might say that there are four aspects to consider for software requirements [Kotonya and Sommerville, 1998; Boehm, 2000, 99]:

- **completeness**: no missing requirements.
- **consistency**: no mismatches among the requirements.
- **traceability**: each requirement can be verified.
- **testability**: each requirement is specific enough as the basis for a pass/fail test for the end product.

These four aspects might be hard to satisfy. As it sometimes is, the users who specify software requirements may not be experienced with the domain of the product to be developed. They might claim, ”I don’t know how to tell you, but I’ll know it when I see it.” This is known as the IKIWISI principle. [Boehm, 2000, 99]. Because of these reasons, software prototypes emerged as a way to deal with software specifications. Generally, two types of software prototypes exist: evolutionary and throw-away prototypes [Sommerville, 2006]. Evolutionary prototyping is an approach for software development where an initial prototype is refined in iterative manner until the prototype satisfies the requirements for the final system. Throw-away prototypes are considered as temporary aids for
discovering requirements problems. The actual system is developed with some other method but based on the prototype. These two approaches are very different from their objectives: an evolutionary prototype should deliver a working system to end users. On the other hand, throw-away prototypes are tools for validating or deriving requirements.

Figure 2.2 Prototyping. [Pressman, 1994, 27]

Oxford English Dictionary defines iterative as making repeated use of a mathematical or computational procedure, applying it each time to the result of the previous application. Prototyping is an iterative software development process. [Pressman, 1994, 27] Prototyping differs from Royce’s model in that sense that the software requirements are to be discovered with succeeding iterations. Royce emphasized the validation of requirements before progressing to more detailed design levels. This assumption is loosened in prototyping because misunderstandings and specification errors can be corrected by further refining the prototype. The six stages of prototyping are depicted in Figure 2.2. Firstly, the early requirements should be gathered and refined. A quick design and prototype based on the early requirements should be provided. The process starts from the beginning after gathering feedback from customer evaluation, and when a prototype is refined as an engineered product. The prototyping process can be iterated as many times as feasible and necessary.
2.1.3   Spiral model

The *spiral model* has characteristics of both linear and iterative process models. [Boehm, 1988, 64] The radial dimension in Figure 2.3 represents the cumulative cost. The progress is indicated by the angular dimension when completing each cycle of the spiral. The underlying concept is that each cycle includes the same sequence of stages. In this sense, the spiral model is an iterative refinement to Royce’s model.

![Figure 2.3 Spiral model of the software process.][Boehm, 1988, 64]

Boehm [1988, 64] emphasizes the risk-driven nature of the spiral model. The spiral gets started by the hypothesis that a particular problem could be improved with a particular software. Thus, a spiral model is a risk assessment to test if a particular problem can really be solved. In case a project is not able to meet its goals, a spiral model development can be terminated. Otherwise, the project should end with an installation of new or modified software. It is necessary to review a cycle after completion and plan the next cycle, if a project is to be continued. This risk-driven approach in the spiral model may be able to better meet the risks in requirements management. The risks in early software requirements are evident in Royce’s model as discussed in Section 2.1.1.
The spiral model does not necessarily require prototyping based approach for requirements management. Boehm [1988, 65] concludes that any appropriate mixture of a specification-oriented, prototype-oriented, simulation-oriented, or other software development approach can be included in the specification stages.

### 2.2 Software development methodologies

A *methodology* is a method of developing information systems, with identified phases and sub-phases, recommended techniques to use and recommendations about planning, management, control and evaluation of a development project.

(Berki et al., 2004)

According to the Merriad-Webster dictionary *methodology* is a series of related methods or techniques. A method is a systematic procedure that contains techniques and tools [Berki et al., 2004, 269]. Cockburn [2002, 115] argues that all organizations have a methodology: it is how they do business. A software development methodology can be described as the actions an organization does regularly to develop software. More precisely, a methodology can be described as the conventions an organization has agreed. What makes methodologies complicated and sometimes intangible is their nature as social constructions. Social constructions imply that people have collectively agreed to behave as certain rules and conventions exist. Some of these rules may be hidden or tacit knowledge. This murky sort of knowledge is possessed without the people being aware of the value of their knowledge.

Software development processes do not necessarily explain all aspects of a methodology. Royce’s model (Section 2.1.1) and the spiral model (Section 2.1.3) do not explain some methodological constructs such as teams, roles or tools. In fact, Cockburn [2002, 115] lists 13 elements for a methodology (Figure 2.4). These elements include *process, milestones, team values, quality, activities, teams, products, techniques, roles, standards, tools, skills and personality*. What the elements actually include vary per organization basis. In the example organization of Figure 2.4, the standards include object-oriented techniques such as C++ and UML modeling. It is worth noticing the elements of a methodology can be applied to any organizational or team based activity, not limiting to software development.
Because methodologies are social constructions, why study them at all? Cockburn [2002, 170] provides some reasons. If new people are introduced, it is helpful to have something available so they can adapt easier to the organization. Secondly, sometimes people need to be replaced. Again, it is helpful to have their roles and responsibilities explicitly expressed. Furthermore, a methodology can also help to make progress of a project more visible, controllable, and manageable. This implies that a methodology can be useful for the understanding and the communication of stakeholders.

A number of software development methodologies exist. Agile software development is explained in Section 2.2.1. In addition, in Section 4.1.3 is introduced a project model for web application development. This project model resembles agile methodologies (such as Scrum [Larman, 2003] and XP [Beck, 2004]) and also some other well-known models such as RUP [Kroll et al., 2003].

2.2.1 Agile software development

According to Oxford English Dictionary agile means the ability to move quickly and easily. Agile software development refers to software development methodologies which claim to be "agile". Agile methods tend to emphasize their light methodology weight although no agreement on the concept of agile actually exists [Abrahamsson et al., 2002, 7]. The term agile software development was born by a group of software practitioners and consultants in 2001. Cockburn [2002, 213] calls the group "advocates of lightweight development". First, appeared the term
"lightweight development" but the agreement was settled on the term agile. They coined the four values of what is called agile software development:

- **Individuals and interactions** over processes and tools.
- **Working software** over comprehensive documentation.
- **Customer collaboration** over contract negotiation.
- **Responding to change** over following a plan.

Abrahamsson et al. [2002, 8] conclude "despite the high interest in the subject, no clear agreement has been achieved on how to distinguish agile software development from more traditional approaches”. However, some points have been made of the methodological foundation of agile software development. Abrahamsson et al. [2002, 17] argue that software development is agile when it is (1) **incremental**, (2) **cooperative**, (3) **straightforward**, and (4) **adaptive**. By the first one we mean that the methodology favors small software releases with rapid cycles. The second value, cooperation, prefers close customer and developer communication with rich communication channels. Third value states that the methodology itself should be easy to learn and well documented. The last value refers to the ability to make last moment changes.

In fact, some of these methodological foundations can be found from some "non-agile" methodologies. The spiral model emphasizes the risk-driven, iterative approach in system development as opposed to large, monolithic software releases [Boehm, 1988]. In 1980s, the Japanese manufacturing industry tried to find new ways to cut down the time to develop new products. Takeuchi and Nonaka [1986, 137] introduced "the new new product development game" which according to them had six characteristics: build-in instability, self-organizing project teams, overlapping development phases, "multilearning", subtle control, and organizational transfer of learning.

As noted, an agile software development methodology should be cooperative and straightforward. However, the aspects of organizational behavior is something which is often left as secondary abstraction in process models [Cain et al., 1996, 3]. Some aspects of organizational behavior are modeled in organizational patterns: the study of the patterns focuses on discovering the social networks and recurring themes in an organization [Cain et al., 1996, 9]. Coplien and Harrison [2004] argue that organizational patterns are a significant contributor to development of agile methodologies.
2.3 Different views of quality

"Quality is the degree of excellence" is the definition of the Oxford English Dictionary for quality. However, quality is an elusive term [Berki et al., 2004]. Therefore, further elaboration is needed. International Standards Organization provides a formal definition of quality: the totality of features and characteristics of a product or service that bear on its ability to satisfy specified or implied needs [ISO, 1986]. In other words, quality is products’ and services’ ability to fulfill its function. The functionality is achieved through the features and characteristics of the product. [Duggan and Reichgelt, 2006, 58] Because multitude of definitions of quality exists, it can be useful to recognize the different aspects of quality definitions. We do this by following Garvin’s [1984] quality perspectives.

Transcendental approach

The transcendental approach resembles the definition of the Oxford English Dictionary. According to this view, quality is something perceived with "innate excellence". Quality is absolute and universally recognizable, although this view claims that quality cannot be defined precisely. Quality is a simple, not analyzable property, which we can learn only through experience.

Product-based approach

According to the product-based approach, differences in quality reflect the differences in the quantity of some attribute possessed by a product. This view states that quality is something which can be exactly measured with certain variables. Based on this view, we could define a number of variables such as the number of features in a software to define the quality. It also follows from this view, that it is possible to rank products based on their quality. For this reason, some mathematical models adapted the product-based view as a definition of quality.

User-based approach

The user-based approach states that quality is highly subjective. It can be said according to this view that "quality lies in the eyes of the beholder." Consumers are assumed to have their individual preferences. Those goods and services which best satisfy their needs are the ones having the highest quality. The user-based approach has some definitional problems. Firstly, how to distinguish quality from those properties which simply maximizes consumer satisfaction. Secondly, how to aggregate varying individual preferences as an overall quality of a product or
service [Duggan and Reichgelt, 2006, 61].

Manufacturing-based approach

The manufacturing-based approach concentrates on the engineering and manufacturing practices. Manufacturing-based views identify quality as the degree of satisfying requirements. Excellence is synonymous to conforming to design and specifications made for the manufacturing process. Quality is internal and not related to end-users. Likewise, the view states that quality should simplify engineering and increase production control. On the design side, the emphasis is on enhancing reliability.

Value-based approach

The value-based approach defines quality in terms of costs and prices. A quality product provides performance at an acceptable price. If the price of a product is not acceptable, the product could not be a quality product because it cannot meet high demand.

2.3.1 Software quality

As previously discussed, quality can be defined from many perspectives. Garvin’s quality perspectives may be helpful to focus on different aspects of quality. However, we can further extend the definition of quality to software quality. Pressman [1994, 550] defines software quality in the following way:

Conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software.

In this definition, requirements are the foundation to measure quality. Furthermore, there should be standards to define a set of development criteria to guide the engineering process. Also, the implicit requirements (for example the desire for good maintainability) should be conformed. [Pressman, 1994, 550] However, this definition is hard to measure and observe in practice. McCall [1977] proposed a categorization of factors which may affect software quality. The quality factors are listed in Table 2.3.1.
<table>
<thead>
<tr>
<th><strong>QUALITY FACTOR</strong></th>
<th><strong>CRITERION</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CORRECTNESS</strong></td>
<td>The extent to which a program satisfies its specification and fulfills the customer’s mission objectives.</td>
</tr>
<tr>
<td><strong>RELIABILITY</strong></td>
<td>The extent to which a program can be expected to perform its intended function with required precision.</td>
</tr>
<tr>
<td><strong>EFFICIENCY</strong></td>
<td>The amount of computing resources and code required by a program to perform its function.</td>
</tr>
<tr>
<td><strong>INTEGRITY</strong></td>
<td>The extent to which access to software or data by unauthorized persons can be controlled.</td>
</tr>
<tr>
<td><strong>USABILITY</strong></td>
<td>The effort required to learn, operate, prepare input, and interpret output of a program.</td>
</tr>
<tr>
<td><strong>MAINTAINABILITY</strong></td>
<td>The effort required to locate and fix an error in a program.</td>
</tr>
<tr>
<td><strong>FLEXIBILITY</strong></td>
<td>The effort required to modify an operational program.</td>
</tr>
<tr>
<td><strong>TESTABILITY</strong></td>
<td>The effort required to test a program to ensure that it performs its intended function.</td>
</tr>
<tr>
<td><strong>PORTABILITY</strong></td>
<td>The effort required to transfer the program from one hardware and/or software system environment to another.</td>
</tr>
<tr>
<td><strong>REUSABILITY</strong></td>
<td>The extent to which a program (or parts of a program) can be reused in other applications – related to the packaging and scope of the functions that the program performs.</td>
</tr>
<tr>
<td><strong>INTEROPERABILITY</strong></td>
<td>The effort required to couple one system to another.</td>
</tr>
</tbody>
</table>

Table 2.1: Software quality factors. [McCall et al., 1977]

McCall’s software quality factors can be split into three subsets. Firstly, some of these quality factors concerns with the requirements specification. As an example, consider the factor *portability* which should be addressed in the software requirements. Secondly, a sort of quality factors are cultural. These factors are not sometimes explicitly stated but expected by the customer. An example of these kind of factors is *usability*. Thirdly, some of the quality factors may be of interest to the developer but not directly to the customer. For example, sometimes *reusability* does not provide value for the customer. [Pressman, 1994, 555]

It can be argued that the division of quality factors to requirement, cultural and development factors is not definite. The quality categories definitely overlap. Depending on the project and its circumstances, some quality factors might
belong to another quality category. For example, usability might be regarded such important factor that it is specified in the requirements. Maintainability is considered as a development factor. Sometimes the case might be that the customer is responsible for the further development of the software. In this case, it is highly likely, that maintainability would be placed in software requirements. [Pressman, 1994, 557]

Next, we proceed further to inspect how software quality can be built and checked with testing. Software testing can be considered as a foundation for quality assurance.

### 2.4 Software testing

The purpose of software testing is to improve the quality of software. We could say, that testing gives us the confidence that the software will work as it should in its intended environment. [Fewster and Graham, 1999, 3] Software testing is not the only approach to improve software quality but, perhaps, one of the most important ones. Inadequate software testing has its effects. Between June 1985 and January 1986 a computer-controlled radiation therapy machine massively overdosed six people [Leveson, 1995]. The accidents were identified to happen because of a software bug was not found in testing. Also, more recently, The Mars Climate Orbiter was lost when it entered the Martian atmosphere. An investigation found out that the software of the orbiter failed to use metric units and crashed on landing. The software was tested using English units. [Leveson, 2004] These two examples underline the importance of software testing. In fact, it is not unusual that over 30% of project efforts are spent on testing. [Pressman, 1994, 609; Royce, 1999, 12]

Myers [1979] states a number of rules which can be considered as testing objectives: (1) Testing is a process of executing a program with intent of finding an error. (2) A good test case is one that has a high probability of finding an as yet undiscovered error. (3) A successful test is one that uncovers an as yet undiscovered error. One could assume that a successful test case is the one that does not find any errors! In fact, the testing objectives differ quite a lot from this common assumption. A conclusion could be that testing should uncover different classes of errors with minimum amount of time and effort. However, as much feedback testing produces, it is important to remember that testing cannot show the absence of defects. Testing can only show that software defects are present. [Pressman, 1994, 611]
Software testing is a subset of *verification* and *validation*. Verification refers to the set of activities that ensure that the software correctly implements a specific function. Validation refers to a different set of activities that ensure that the software has been built upon customer requirements. [Pressman, 1994, 646] Boehm [1981, 37] states this with two questions: *Are we building the product right?* (Verification), *Are we building the right product?* (Validation).

### 2.4.1 V-model

Testing is sometimes considered something which is done after software has been written. The argument is that how can you test something that does not exist? [Fewster and Graham, 1999, 6] Of course, executing tests requires at least some parts of the software to be finished. The V-model of software development [Fewster and Graham, 1999, 7] in Figure 2.5 illustrates *when* testing activities should take place. The view that software should be tested after it is written would imply that only acceptance tests should be performed. There is more to testing than that. The V-model shows the corresponding testing activities related to each development stage. These development stages can be found from various software process models such as Royce’s linearly progressing model and Boehm’s iterative spiral model. The V-model’s testing activities can be defined as follows.

![Figure 2.5](image-url) The V-model showing early test design. [Fewster and Graham, 1999, 7]
Unit testing

Unit testing directs verification focus on the smallest unit of software design, that is the module. [Pressman, 1994, 652] Some source code level problems, which may be covered with unit tests, include (1) improper or inconsistent typing, (2) erroneous initialization or default values, (3) incorrect (misspelled or truncated) variable names, (4) inconsistent data types, and (5) underflow, overflow, and addressing exceptions [Pressman, 1994, 653]. Because a module is not a stand-alone program itself, a driver and/or stub software must be developed for executing test cases [Pressman, 1994, 654]. The unit tests can be written after a module has been written. Another approach is to write unit tests first which is the approach taken in some software development methodologies, such as Test-Driven Development [Beck, 2002]. A number of frameworks exist for writing unit tests. Some of these include JUnit [Massol and Husted, 2003], NUnit [Hunt et al., 2007] and TestNG [Beust and Suleiman, 2007]. The purpose of these frameworks is to reduce overhead: less boilerplate code is required for executing unit tests.

Integration testing

Here, it is enough to define integration as a combination of software modules; a better definition is introduced in Section 3.1. Integration testing is a testing activity where the individual software modules are tested together. [Pressman, 1994, 655] Even if the individual modules are unit tested and seem to work, a combination of modules can introduce problems. Integration testing is a technique for constructing the program structure. At the same time tests should be conducted to uncover integration errors. [Pressman, 1994, 656] Two approaches can be taken for integration testing: top-down [Pressman, 1994, 656] and bottom-up integration testing [Pressman, 1994, 658]. Top-down integration begins by compiling modules downward through the control hierarchy, beginning with the main control module or main program. On the other hand, bottom-up integration begins with atomic modules. Low-level modules are combined into clusters and the clusters are tested. The clusters are combined until the top-level or the main control module is reached.

System testing

Software is an element of a larger context, that is, a computer-based system. Software is incorporated with other system elements such as hardware or data communications links. System testing is conducted with a series of different
tests whose purpose is to fully exercise the computer-based system. [Pressman, 1994, 665] Some of the system testing activities include recovery testing, security testing, stress testing and performance testing.

Acceptance testing

Acceptance testing concentrates on the end-user point of the software. The purpose of acceptance tests is to enable the customer to determine if all software requirements are satisfied. The time to conduct acceptance testing is when all functionality specified by the requirements is done. There are two possible outcomes from acceptance testing: (1) the function and performance characteristics conform to specification and are accepted, or (2) the software does not fulfill its intended functionality, thus a list of deficiency list is created. [Pressman, 1994, 663]

2.4.2 Testing approaches

There are two fundamental approaches for testing any engineered product. The first approach is to test the product by its designed functionality without paying attention to the internal structure and implementation of the product. The second approach is to concentrate on testing the internal operation of the product. That is, that the internal operations perform according to specification. [Pressman, 1994, 612] The first approach is called black box testing and the second white box testing. Next, we state these approaches more precisely.

Black box testing

As stated previously, black box testing pays little attention to the internal structure of the software. Black box testing attempts to find errors in the following categories: (1) incorrect or missing functions, (2) interface errors, (3) errors in data structures or external database access, (4) performance errors, and (5) initialization and termination errors. [Pressman, 1994, 631] Black box testing tends to be applied in the later stages of testing [Pressman, 1994, 631]. That is, in the system testing and acceptance testing stages. A number of black box testing methods exist, such as equivalence partitioning, boundary value analysis, cause-effect graphing techniques, and comparison testing [Pressman, 1994, 632, 633, 634, 637].
White box testing

The test cases for white box testing are derived from the internal structure of the software, as previously stated. Using white box testing, one can derive test cases that (1) guarantee all independent paths within a module have been exercised at least once, (2) exercise all logical decisions on their true and false sides, (3) execute all loops at their boundaries and within their operational bounds, and (4) exercise internal data structures to ensure their validity. [Pressman, 1994, 613–614] White box testing is applicable in unit testing, integration testing and system testing. However, white box testing is not suitable for acceptance testing because acceptance testing should uncover missing requirements. Naturally, a testing approach which focuses on testing the existing functionality cannot detect missing functionality. White box testing methods include basis path testing, condition testing, data flow testing, and loop testing [Pressman, 1994, 615, 625, 628, 630].

2.4.3 Test automation

Test automation is an approach to automate otherwise manually conducted testing. The tests are run with a test execution tool and the actual outcomes are compared with the expected outcomes. Executing automated tests can be more effective than manual testing in some cases because the tests can be rerun with little or no effort at all. Some manual test processes can be replaced with automated tests but it is naive to expect that the move to automated testing would be a straightforward process. In fact, it is more expensive to automate a test than to perform it once manually. It is also worth noticing that automating a test itself is not a mean; as we stated before "testing is a process of executing a program with intent of finding an error". It does not mean that once a test is automated it is more capable of finding errors. Although, in some cases, automated tests can be more effective. [Fewster and Graham, 1999, 5] Bach collected a list of common false assumptions which may be helpful to notice in the context of test automation [Bach, 1996]. These are listed below:

1. Testing is a "sequence of actions".

At the first glance, testing may seem like a sequence of actions. Humans are good in detecting hundreds of problem patterns and instantly distinguish them from harmless anomalies. This implies that good testing is an interactive cognitive process where the tester adapts the testing activities based on software’s behavior. It is possible that automated tests report problems that are not significant nor
problems at all.

2. *Testing means repeating the same actions over and over.*

A test case is unlikely to find a defect if it does not find a defect on the first run. If there is variation in the test cases, defects are more likely to be found.

3. *We can automate testing actions.*

Not all testing activities or types of testing can be automated. Bach argues that the hardest part of automation is interpreting test results. For example, automatically noticing significant and non-significant defects from a graphical user interface can be very hard.

4. *An automated test is faster, because it needs no human intervention.*

All automated tests require human intervention because the test results need to be diagnosed and broken tests must be fixed. It can be tedious to make a complicated test suite to run without problems. Common culprits include changes to the software being tested, memory problems, file systems problems, network problems and defects in the test tool itself.

5. *Automation reduces human error.*

Some errors can be reduced but automation can also systematically ignore defects in the software. Bach reported a case where the tests systematically reported success regardless of the test status.

6. *We can quantify the costs and benefits of manual vs. automated testing.*

Bach argues that manual and automated testing are two different processes rather than two different ways to execute the same process. This results to different kinds of defects uncovered. It can be hard to compare manual and automatic testing in terms of cost or the number of defects founds.

7. *Automation will lead to "significant labor cost savings".*

Test automation is costly in the short run because at least time need to be invested to test tools. In the long run, the test infrastructure needs to be maintained which causes further costs. It can be hard to argue if the labor costs can decrease with automated testing; in fact, test automation may require initially more labor.
8. Automation will not harm the test project.

There need to be a clear understanding what is going to be automated and which parts of the tests are not covered by automation. Automating chaos just gives faster chaos [Fewster and Graham, 1999, 11]. According to Bach, there can be a fear to maintain the test suite. As a result, the test suite can be a growing burden to the project.

Keeping these shortcomings in mind, we can find promises of successful automated testing according to Fewster and Graham [1999, 9].

1. Run existing (regression) tests on a new version of a program.
It should be possible to run automated tests on a new version of software with little effort. This may be beneficial if new software versions are released often. The regression tests can be used to check that existing tests can be successfully run with the new version.

2. Run more tests more often.
The ability to run tests more often will lead to greater confidence in the system. It is possible to run tests more often because the time required to execute tests is less than with manual tests.

3. Perform tests which would be difficult or impossible to do manually.
There are some testing scenarios which are difficult or impossible to test manually. An example is a test with 200 simultaneous users which might be difficult to arrange in practice but can be simulated with a testing software. This fact does not conflict with the false assumption of "we can automate testing actions". The fact states that in some cases automatic testing is a better approach than manual testing and, likewise, automatic testing cannot replace manual testing in some cases.

4. Better use of resources.
Automating menial or boring tasks, such as repeatedly entering the same test inputs, gives greater accuracy as well as improved staff morale. The testing resources can be used to design better test cases to be run.

5. Consistency and repeatability of tests.
The non-creative nature of automated tests can be also seen as a strong point.
Tests that are repeated automatically will be executed exactly with the same inputs. This gives consistency which might be hard to achieve manually.

6. **Reuse of tests.**

The effort spent on designing and building the tests can be distributed over many executions of those tests. The efforts on test automation may be seen generally beneficial and improve organization's software development process.

7. **Earlier time to market.**

If the testing time can reduced with automated tests, it might be possible to reduce the time to market. This is subject to other factors, such as the ability of developers to fix defects.

8. **Increased confidence.**

If the automated tests detect defects effectively, there can be greater confidence that there will not be unpleasant surprises when the system is released.

2.4.4 Approaches for automating test case design

As stated previously, the basis for test automation can be the manual test cases. However, a tedious part of test automation can be the construction of test cases. Why not to automate test case design? There are, indeed, approaches for automating test case design. These approaches can be classified to three categories. [Fewster and Graham, 1999, 19]

*Code-based test cases*

Code-based test input generation tools generate test inputs by analyzing the structure of the software code itself. Code-based test case design cannot tell whether the outcomes produced by the software are correct. [Fewster and Graham, 1999, 19] However, some automated test tools can derive useful information based on the code. For example, static analyzers such as FindBugs [FindBugs, 2009] can uncover logical errors based on the source code structure.

*Interface-based test cases*

Interface-based test input generation tools generate test cases based on some well-defined interface, such as a web application or a GUI. An interface-based approach can find some type of defects and identify expected outputs. [Fewster
and Graham, 1999, 20] As an example, the test paths for a web application could be generated automatically [Miao et al., 2008].

**Specification-based test cases**

Specification-based test input generation tools can generate test inputs and expected outputs by a specification. Of course, the specification must be in a form which is analyzable to the testing tool. The specification may contain business rules or technical data such as states and transitions. [Fewster and Graham, 1999, 21]

### 2.5 Summary

Software development processes and methodologies can be utilized as a systematic technique to organize software development activities in an organization. Currently, a number of processes and methodologies exist such as Royce’s model, the spiral model, and agile software development. Processes and methodologies can be seen as an approach to improve the software engineering activities. That is, higher quality software should be produced as a result.

A multitude of definitions for software quality exists. This is because software quality is a multifaceted subject. Although no definite agreement on the concept of software quality exists, testing can be seen as an approach to build and check software quality.

In the next chapter, the continuous integration technique is shown as a practice which can be utilized in a software process. By integrating automated functional testing activities to continuous integration an organization can get rapid feedback on the quality of the software under development. This is shown to be a practice, which improves software quality.
3 REVIEW OF THE LITERATURE

3.1 Software integration


- **Systems integration** is the practice of combining the functions of a set of subsystems, be it software, hardware or both, to produce a single, unified system that satisfies some need of an organization. [Kuhn, 1990]

- **Software integration** is the practice of assembling a set of software components/subsystems to produce a single, unified software system that supports some need of an organization. [Stavridou, 1999, 3]

- **Tool integration** is the practice of combining a set of software development tools to produce a single, unified software development environment. [Stavridou, 1999, 3]

![Figure 3.1 Progress profile of a conventional software project. [Royce, 1999, 12]](image)
a project progress in a linear fashion (requirements, design, coding, integration, testing) such as in the Royce’s model. Progress is defined as percent coded, that is, demonstrable in its target form [Royce, 1999, 12]. What usually results after the individual software components are finished is a set of integration problems. Royce [1999, 12] calls this phenomena “integration nightmare” due to unforeseen implementation issues and interface ambiguities. Usually the original target date for project completion is missed and the software design needs changes because of integration problems. The integration and test stage can account as far as 40% of expenditures. This is depicted in Table 3.1. Are there any means to address this late risk resolution of integration? Indeed, there are! In the next section, we suggest *continuous integration* as a way to mitigate integration risks.

<table>
<thead>
<tr>
<th>Activity</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Management</td>
<td>5 %</td>
</tr>
<tr>
<td>Requirements</td>
<td>5 %</td>
</tr>
<tr>
<td>Design</td>
<td>10 %</td>
</tr>
<tr>
<td>Code and unit testing</td>
<td>30 %</td>
</tr>
<tr>
<td>Integration and test</td>
<td>40 %</td>
</tr>
<tr>
<td>Deployment</td>
<td>5 %</td>
</tr>
<tr>
<td>Environment</td>
<td>5 %</td>
</tr>
<tr>
<td>Total</td>
<td>100 %</td>
</tr>
</tbody>
</table>

Table 3.1: Expenditures by activity for a conventional software project. [Royce, 1999, 13]

### 3.2 Continuous integration

*Continuous integration* [Duvall *et al.*, 2007] as a term is a bit vague. A better term would be "continuous software integration". It is worth noticing that even that would not be technically correct because the word *continuous* has the meaning of "without interrupting" or "forming a series with no exceptions or reversals" according to Oxford English Dictionary. What continuous integration means is that software integration is performed with a certain schedule. For example, we could say that integration is performed several times a day.
There is a need for some new terms with continuous integration. An integration cycle, or more precisely the results of an integration cycle are called a build. Here, we use the definition by Duvall et al. [2007, 4] which has much in common with how we defined software integration:

A build is much more than a compile (or its dynamic language variations). A build may consist of the compilation, testing, inspection, and deployment – among other things. A build acts as the process for putting source code together and verifying that the software works as a unit with high cohesion.

It is easier to illustrate continuous integration with a typical continuous integration process. In Figure 3.2 is an example process. The elements in the process are version control repository, CI server, build script and feedback mechanism. We introduce each of these elements before commenting on the overall CI process.

---

**Figure 3.2** The components of a CI system. [Duvall et al., 2007, 5]

---

**Version control system (VCS)**

Version control system is a prerequisite for continuous integration [Duvall et al., 2007, 7]. It combines procedures and tools to manage different versions of configuration objects that are created during the software engineering process [Pressman, 1994, 711]. These configuration objects may contain source code, documents or data required for a build. The data of a VCS is stored in one or many version control repositories (or simply a repository). A VCS must handle the data in
that manner that developers have a way to know the exact differences between different versions of objects stored in a repository.

A number of version control systems exist. Concurrent Versions System (CVS) [Purdy, 2003], Subversion (SVN) [Pilato et al., 2008] and Git [Swicegood, 2008] are some of these systems. There should be a set of operations available for a VCS. Committing means that a developer checks in a new version of a configuration object. If the object does not conflict with the previous version, it is added as a new version. Also, there should be a branch called head or trunk, which might be called with a different name with different VCSs. This mainline branch contains the newest or the latest version of committed objects. Typically, a continuous integration system is executed against this mainline branch to integrate the latest changes.

**CI server**

The main purpose of a CI (continuous integration) server is to execute a build whenever a change is committed to the version control repository. Typically, changes are checked every few minutes. If there are new changes, the CI server will retrieve objects from the VCS and execute a build. CI servers usually provide a dashboard, where build results can be seen. In a technical sense, a CI server is not strictly required for continuous integration but can leverage the adaptation process because less customization and programming is needed. [Duvall et al., 2007, 8] Some options for a CI server include Apache Continuum [Apache, 2009], Bamboo [Atlassian, 2009], CruiseControl [CruiseControl, 2009] and Hudson [Hudson, 2009].

**Build script**

Build script is a script or a set of scripts for compiling, testing, inspecting and deploying software. [Duvall et al., 2007, 10] A build script can be written with a programming language or a dedicated build tool. Some of the dedicated build tools include GNU Make [Stallman and McGrath, 2002], Apache Ant [Loughran and Hatcher, 2007] and NAnt [Holmes, 2005]. The advantage of these tools is that they often do not require programming for achieving build automation. For example, the build scripts of Apache Ant are XML files. Some of the tools include further automation capabilities such as dependency management. That is, a dependent component can be automatically included as a part of a build.
**Feedback mechanism**

One of the key features of continuous integration is a feedback mechanism. There are two outcomes for a build: it can be successful or it can fail. A build is failed if it contains errors. [Duvall et al., 2007, 10] Typical reasons for failed builds include compilation errors, unit tests failures, static analysis failures and server problems [Miller, 2008, 289]. A feedback mechanism should inform the specified project members about build failures and successes. If a build is failed, usually the feedback mechanism tries to inform on the reason of failure. Some CI systems can also notify only the developer, who originally made the change that broke the build [Atlassian, 2009]. Typically, feedback is sent with an e-mail, Short Message Service (SMS) or Really Simply Syndication (RSS). Immediate feedback can help the development team to react to build failures promptly. [Duvall et al., 2007, 10]

**CI process**

Duvall et al. [2007, 12] list four requirements for a continuous integration process. These requirements are (1) a connection to a version control repository, (2) a build script, (3) some sort of feedback mechanism (such as e-mail), and (4) a process for integrating the source code changes (manual or CI server). Based on these requirements, we can introduce a process for continuous integration. A typical CI process is described in Figure 3.2.

1. First, a developer commits code to the version control repository. Meanwhile, the CI server polls the repository for changes (e.g., every few minutes).

2. Soon after a commit occurs, the CI server detects that changes have occurred in the version control repository. The CI server retrieves the latest copy of the code from the repository and then executes a build script to integrate the software.

3. Feedback is generated by the CI server to notify the specified project members about the build results.

4. The CI server continues to poll for changes in the version control repository.

A simple cost-benefit analysis might not be the best way to compare the advantages of continuous integration and other integration approaches. However, Miller [2008] gathered data of using continuous integration from 551 builds in Microsoft. Miller’s [2008, 291] cost-benefit analysis revealed that the overhead for
a project for CI was 267 hours, 7% of the total effort. Compared to a hypothetical heavyweight check-in process for that team, the estimated overhead would have been 464 hours. Beyond this hypothetical cost-benefit analysis, Miller found some other reasons to favor CI. Those reasons include (1) frequent check-ins makes integration effort easier (especially with multi-site teams), (2) fewer merge conflicts are caused, (3) project members have a better visibility of the current state of the code, (4) code reviews take less time, (5) the code base can be stabilized in the release stage with less effort [Miller, 2008, 291–292].

3.2.1 Continuous testing

As earlier stated, a minimum requirement for a continuous integration process is the capability to integrate the source code changes. Thus, the main feature of continuous integration is to produce an integration build. Is there any reason not to further automate the build process? Indeed, there is empirical evidence [Cannizzo et al., 2008; Miller, 2008] that automated testing can add more value to a CI process. Continuous testing means that a set of tests are executed per build basis and if the tests fail, the build also fails. This can provide additional quality feedback for a project. However, the tenets of test automation apply: test automation itself cannot guarantee software quality.

We can classify builds based on the type of testing they embody. (1) A commit build executes unit tests, (2) an integration build is for integration tests, and (3) a functional build is for executing system tests and/or acceptance tests. [Duvall et al., 2007, 141] One could argue, if it is necessary to limit testing activities to certain builds. Suppose that all testing activities (unit, integration, functional, acceptance) are not run per commit basis, does this mean that some defects are not found? Unfortunately, the "testing weight" tends to increase when moving from unit to integration and functional to acceptance testing. This means that the tests take longer to run. In some types of testing (such as performance testing) it is necessary to execute the tests for days [Cannizzo et al., 2008]. The problem of increased running times is the loss of immediate feedback. This immediate feedback is necessary for a development team to know the immediate quality of a build, that is, if the code compiles and is ready for further testing. In this sense, it is useful to have different classes of builds.

Based on the type of a build (commit, integration, functional) we can define a schedule for running the build. Commit builds should be executed per commit basis providing immediate feedback. Integration builds can be run as a
dependent build for a commit build. This implies that a successful commit build yields an integration build, which is assumed to complete within longer time. Functional builds can be run during nights or idle times. For example, Cannizzo et al. [2008] describe an approach where performance and robustness builds are executed during weekends.

Some CI testing frameworks are introduced in Table 3.2. The frameworks include JUnit [Massol and Husted, 2003], NUnit [Hunt et al., 2007] and TestNG [Beust and Suleiman, 2007] for unit testing, DbUnit [DbUnit, 2009] for integration testing, JWebUnit [JWebUnit, 2009] for system testing and Selenium [Selenium, 2009] for acceptance testing. All these testing frameworks can be utilized as a part of continuous testing. Other types of tools that can be utilized include code coverage tools and static analyzers such as FindBugs [FindBugs, 2009].

<table>
<thead>
<tr>
<th>Testing activity</th>
<th>Testing objective</th>
<th>Example frameworks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit testing</td>
<td>Code</td>
<td>JUnit, NUnit and TestNG</td>
</tr>
<tr>
<td>Integration testing</td>
<td>Design</td>
<td>DbUnit</td>
</tr>
<tr>
<td>System testing</td>
<td>Functions</td>
<td>JWebUnit</td>
</tr>
<tr>
<td>Acceptance testing</td>
<td>Requirements</td>
<td>Selenium</td>
</tr>
</tbody>
</table>

Table 3.2: Testing frameworks can be used as a part of continuous testing to verify different testing objectives. The frameworks here are exemplary and many others can be found. The testing objectives are from the V-model, previously introduced.

3.2.2 Atlassian Bamboo

Atlassian Bamboo is a continuous integration server. The main view of Bamboo can be seen in Figure 3.3. Currently, in the version 2.2.1, Bamboo supports CVS, Subversion and Perforce version control systems. A source repository plugin module may be used for supporting of other types of version control systems. For different classes of builds (in Atlassian terms build plans) different sources or different version control systems could be defined. For example, there could be the build plan X which is located in CVS, and the build plan Y, which is located in Perforce. [Atlassian, 2009]
The build script support in Bamboo is implemented with builders. A builder is an external program to Bamboo, which builds a certain build. Bamboo supports multiple builders. Currently, Ant, Maven, NAnt, devenv.com, and custom scripts/commands are supported. Depending on the builder type, the configuration for the builder varies. For example, the Ant builder requires the location of the Ant script (.build.xml) specified.

Build notifications can be defined for completed builds. That is, project members can be notified of different build outcomes. These outcomes include all completed builds, failed builds, and commented builds. Bamboo can automatically send build notifications with email or an instant message. Extensible Messaging and Presence Protocol (XMPP) based instant messaging services such as Google Talk are currently supported in Bamboo.

![Atlassian Bamboo](image)

**Figure 3.3** Atlassian Bamboo.

### 3.3 Web application security testing

#### 3.3.1 Web applications

In 1991, Timothy Berners-Lee released the first World Wide Web (WWW) server. The World Wide Web (or simply the web) was the first widely adapted implementation of hypertext [Bush, 1945] – a document where a set of documents can
refer to each other [CERN, 2008]. The web can be accessed with the Hypertext Transfer Protocol (HTTP). The version 1.0 of the HTTP protocol was proposed by Berners-Lee et al. [1996]. From here on, we simply call WWW the web and hypertext documents web pages.

The content of web pages can be generated in two ways. A static web page always comprises the same information in response to all requests from all users. A dynamic web page provides "interactive" content. That is, a web page can react to user's input with some specific way. Probably one of the most well-known dynamic web pages is the Google Search [Google, 2009], where a user can search the web with a keyword. Google's search reacts to the keyword provided by the user and displays search results. Dynamic web pages are generated by web applications.

A web application is comprised of a collection of dynamic scripts, compiled code, or both, that reside on a web, or application server and potentially interact with databases and other sources of dynamic content [Andreu, 2006, 16]. The physical architecture of a web application can be seen in Figure 3.4(a). The architecture has three layers: (1) internet, (2) web tier, and (3) data tier. Internet is used as a transport network between a web server and an end-user. Typically, a firewall is used to limit the access to the web server. The web tier is served by a web server which is a software responding to HTTP requests. Currently, the two most popular web servers include Apache HTTP [HTTP, 2009] and Microsoft IIS [IIS, 2009] [Netcraft, 2009]. A web server can include an application server (such as Apache Tomcat [Tomcat, 2009]) for executing web applications. The data tier handles storing and retrieving information. The data tier can be implemented with a database server such as MySQL [DuBois, 2008].

The architecture depicted in Figure 3.4(a) does not capture the real complexities of web applications. Most web applications are complex systems that integrate and exchange data with other systems and store and process data in many different places [Curphey et al., 2006, 32]. A more realistic web application architecture is depicted in Figure 3.4(b) which includes complexities such as interactions with different applications, protocols (for example RPC, SMTP, XML/HTTP) and gateways for transferring data. Although the most visible part from the end user's point of view, it is important to remember the user interface is usually only a part of a system: a large web application typically consists of many back-end systems and databases.

Lowe and Henderson-Sellers [2001] argue that a general distinction should be made between conventional and web systems. The distinctions can be categorized
Figure 3.4 Two views of a web site. [Curphey et al., 2006, 33]
in technical and organizational differences [Lowe and Henderson-Sellers, 2001, 1]. The technical differences include (1) link between business model and architecture, (2) open modular architectures, (3) rapidly changing technologies, (4) emphasis on content, (5) importance of user interface and (6) increased importance of quality attributes [Lowe and Henderson-Sellers, 2001, 2]. The organizational characterizations for web application development include (1) client uncertainty, (2) changing business requirements, (3) short time frames for initial delivery, (4) highly competitive, and (5) fine-grained evolution and maintenance [Lowe and Henderson-Sellers, 2001, 2–3].

3.3.2 Web application security vulnerabilities

A vulnerability is a weakness in a system caused by a flaw in design, a coding error, or incorrect configuration such that execution of a program can violate the implicit or explicit security policy. An example of a security vulnerability is a remote code execution vulnerability in Microsoft Office Powerpoint. US-CERT (United States Computer Emergency Readiness Team) maintains a Vulnerability Notes Database [US-CERT, 2009b]. At the time of writing this thesis the vulnerability in Microsoft Office Powerpoint is the latest one in the Vulnerability Notes Database with the number 627331 [US-CERT, 2009a].

The cause of a security vulnerability can be an implementation bug or a design flaw [Curphey et al., 2006, 33]. This distinction might seem to be overly simplified but it has been found useful [Curphey et al., 2006, 34]. As an example, the PowerPoint vulnerability previously mentioned is an implementation bug. In contrast, storing plain text passwords in a database could be a design flaw.

In this section, we focus on the web application security vulnerabilities. Probably the most common vulnerabilities in web applications are input validation vulnerabilities. In fact, the US National Vulnerability Database showed that in 2008, 33.46% of the vulnerabilities (not limiting to web applications) can be accounted to SQL injection and cross-site scripting vulnerabilities! [CVE, 2009] Next, we define these vulnerabilities in more detail.

**SQL injection vulnerabilities**

SQL (Structured Query Language) is a language designed for the retrieval and management of data stored in a relational database system. A number of web applications handle the data storage with a relational database. SQL injections are based on altering the syntax of an SQL query. By injecting another string
into an SQL query the structure of the query can be changed. For example, a malicious user could change an SQL query to delete data instead of retrieving it with an SQL injection vulnerability. In Listings 3.1, 3.2, 3.3 and 3.4 is constructed a typical SQL injection. The query in Listing 3.1 retrieves the user with the name "john" and the password "doe".

```
SELECT ID, LastLogin FROM Users
    WHERE User = 'john' AND Password = 'doe'
```

Listing 3.1: SQL Injection Step 1.

In Listing 3.2 the SQL query is executed with a hypothetical programming language. The static values "john" and "doe" are replaced with the variables userName and password. During runtime the content of these variables is substituted with user input. This allows to execute the query with a database.

```
sqlQuery = "SELECT ID, LastLogin FROM Users
    WHERE User = "+ userName + " AND Password = "+ password + ""
```

Listing 3.2: SQL Injection Step 2.

Actually, a malicious user could enter any strings as an input to the query, if the user input is not validated. For example, consider the SQL query to be constructed with values shown in Listing 3.3.

```
User: ' OR 1=1 --
Password: 
```

Listing 3.3: SQL Injection Step 3.

When the user input in Listing 3.3 is substituted with the SQL query, the resulting query can be seen in Listing 3.4. The original query in Listing 3.1 gets the user information for a single user as defined by the username and password. The modified query does not adhere the original semantics! Instead, it retrieves the details for a random user and always succeeds. Thus, the authentication with a username and password is effectively passed.

```
SELECT ID, LastLogin FROM Users
    WHERE User = ' ' OR 1=1 -- AND Password = '
```

Listing 3.4: SQL Injection Step 4.
To avoid these kind of vulnerabilities, the user input should always be validated. The consequences of an SQL injection can range from data loss to authentication problems. Often the programming languages and frameworks offer a way to deal with SQL queries in a secure manner [Kals et al., 2006, 1]

Cross-site scripting vulnerabilities (XSS)

Cross-site scripting vulnerabilities are a type of vulnerability which can be exploited by injecting malicious JavaScript into a web application [CERT, 2000]. When a web page with an XSS vulnerability is viewed, the content of the web page seems to originate directly from the web site itself, although the web page contains a "hidden" malicious JavaScript. As a result, the script can access and steal cookies, session IDs, and other sensitive information [Kals et al., 2006, 3].

There are reflected and stored XSS attacks. Reflected attack is the most commonly exploited XSS attack. [Kals et al., 2006, 3] Typically, a reflected attack is exploitable when a web application directly outputs user input without validating or escaping the input. As an example, say there is a search form and the search term is entered as a part of the URL: \texttt{http://search.example.com/?query=\{userinput\}}. Now, the web application displays the user input directly without validating or escaping it (listing 3.5). Consider a string containing JavaScript is entered. Now, the browser executes the JavaScript without the knowledge of the end user!

\begin{verbatim}
<form>
   You searched for:
   <input type="text" value="\{user input\}" />
   <input type="submit" value="Search again" />
</form>
\end{verbatim}

Listing 3.5: A web application outputs user input directly.

The difference of a stored and reflected XSS attack is that a stored attack does not immediately output the injected data. For example, the XSS attack string (a JavaScript string) could be stored in a database and go as output to a web page when the data is later retrieved from the database. Again, the cause for the vulnerability is the lack of proper input validation. Consider the HTML output in Listing 3.6. A user can post messages to an online discussion forum. A malicious user embeds a snippet of JavaScript to the message. When the other users retrieve the message from the discussion forum, they execute the script
embedded in the message. This is a stored XSS attack.

```html
<p>
 This is a message posted by a malicious user.

 Input is not validated properly.
</p>
<script type="text/javascript">

 // Hijack the session information from other users
</script>
```

Listing 3.6: A stored XSS attack.

3.3.3 Web application security assessment tools

The purpose of web application security testing is to find security vulnerabilities from a web application. As we stated before, testing can be conducted with a <em>white box</em> or <em>black box</em> approach. In white box security testing the source code of an application is analyzed to uncover vulnerable lines of code. Black box security testing could be described as a "hacker-like" approach. The tester tries to "break" an application by entering specific inputs such as XSS injection strings. Vulnerabilities are detected by analyzing unexpected behavior or errors indicated by the application. [Kals et al., 2006, 1]

Tool-based approaches for web application security testing has been proposed by Auronen [Auronen, 2002]. Auronen concluded that testing the security of web application with the tools introduced is not enough. The limitations of the tools included the lack of versatility of a human auditor. Also, no white box security testing tools existed during that time [Auronen, 2002, 17]. Currently, a number of white box testing tools exist such as WebSSARI [Huang et al., 2004] and Pixy [Jovanovic et al., 2006]. The fundamental difference of manual testing and test automation was previously presented. Each of these tools can be manually used to aid the tester or as a more grandiose approach to automate testing. If the test automation approach is chosen, it is important to remember the implications of test automation, as previously introduced.

Curphey et al. introduced a categorization of web application security assessment tools, namely <em>source-code analyzers, web application (black-box) scanners, database scanners, binary analysis tools, runtime analysis tools, configuration analysis tools, proxies tools, and miscellaneous tools</em> [Curphey et al., 2006].
We use this categorization to introduce the tools available. In Table 3.3 is an overview of tools fitting into these categories.

<table>
<thead>
<tr>
<th>Tool Type</th>
<th>Example Tools</th>
<th>Life-Cycle Phase</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Source-code analyzers</strong></td>
<td>WebSSARI [Huang et al., 2004], Pixy [Jovanovic et al., 2006], FindBugs [FindBugs, 2009]</td>
<td>Development</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Testing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Pre-deployment</td>
</tr>
<tr>
<td><strong>Web application (black-box) scanners</strong></td>
<td>WAVES [Huang et al., 2003], SecuBat [Kals et al., 2006]</td>
<td>Pre-deployment</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Post-deployment</td>
</tr>
<tr>
<td><strong>Database scanners</strong></td>
<td>AppDetectivePro ([<a href="http://www.appsecinc.com/products/appdetective/">http://www.appsecinc.com/products/appdetective/</a>])</td>
<td>Testing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Pre-deployment</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Post-deployment</td>
</tr>
<tr>
<td><strong>Binary analysis tools</strong></td>
<td>FxCop ([<a href="http://msdn.microsoft.com/en-us/library/bb429476.aspx">http://msdn.microsoft.com/en-us/library/bb429476.aspx</a>])</td>
<td>Testing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Pre-deployment</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Post-deployment</td>
</tr>
<tr>
<td><strong>Runtime analysis tools</strong></td>
<td>Computerware BoundsChecker ([<a href="http://www.compuware.com/products/devpartner/visualc.htm">http://www.compuware.com/products/devpartner/visualc.htm</a>])</td>
<td>Development</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Testing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Pre-deployment</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Post-deployment</td>
</tr>
<tr>
<td><strong>Configuration analysis tools</strong></td>
<td>Desaware CAS/Tester ([<a href="http://www.desaware.com/products/castester/index.aspx">http://www.desaware.com/products/castester/index.aspx</a>])</td>
<td>Development</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Testing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Pre-deployment</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Post-deployment</td>
</tr>
<tr>
<td><strong>Proxies tools</strong></td>
<td>ratproxy ([<a href="http://code.google.com/p/ratproxy/">http://code.google.com/p/ratproxy/</a>])</td>
<td>Testing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Pre-deployment</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Post-deployment</td>
</tr>
</tbody>
</table>

Table 3.3: Tool overview. [Curphey et al., 2006, 36–37]

Source-code analyzers

Source-code analyzers analyze security vulnerabilities from source code with a number of techniques such as data flow analysis [Jovanovic et al., 2006, 2; Curphey...
et al., 2006, 34]. Because of a relatively low overhead of source-code analyzers, these tools can be typically introduced in a continuous integration build without adding too much overhead [Cannizzo et al., 2008].

Web application (black-box) scanners

Web application scanners mimic an attack. These scanners may look for known vulnerable flaws from a web application by scanning certain URLs (such as administrative interfaces). Another approach is to scan the web application structure and try to exploit the application with security test cases. These cases may include tests for vulnerabilities such as SQL injections and XSS vulnerabilities. [Curphey et al., 2006, 38]

Database scanners

Database scanners typically act as SQL clients and perform various queries to analyze the security configuration of a database. Database scanners also verify database users and role memberships against known best practices. [Curphey et al., 2006, 38]

Binary analysis tools

Binary analysis tools can be used to check low-level security vulnerabilities such as buffer overflows without access to the source code [Curphey et al., 2006, 39]. Some of these tools, such as FxCop can be used as a part of a continuous integration build [Miller, 2008].

Runtime analysis tools

Runtime analysis tools can be used as an aid for determining behavior of an application. The tools do not identify any errors but can be used by a tester to uncover application behavior such as calls to certain API (Application Programming Interface) functions. [Curphey et al., 2006, 39]

Configuration analysis tools

Configuration analysis tools perform a static analysis against the configuration files of an application. Examples include evaluating security of a configuration of a web application server. [Curphey et al., 2006, 38–39]

Proxies tools

Proxies tools can be used as a middle layer between a web application and a
tester to debug a web application. Proxies allow the tester to capture HTTP requests or modify them to inspect the behavior of a web application [Curphey et al., 2006, 40].

**Miscellaneous tools**

Miscellaneous tools include tools that do not directly fit into any of these categories such as unit testing frameworks [Curphey et al., 2006, 40].

### 3.3.4 Acunetix Web Vulnerability Scanner

Acunetix Web Vulnerability Scanner (WVS) is a web application scanner as defined in Section 3.3.3. The testing approach of WVS is black box scanning. However, the newest version of WVS (6.1), at the time of writing this thesis seems to have incorporated white box techniques for further refining the scanning results. [Acunetix, 2009] Test cases can be automatically generated from the web application structure. This is done with the crawler component of WVS, which mimics a browser: it recursively follows the links from web pages and adds individual pages as test cases. WVS is installed as a stand alone application. With certain licensing options unlimited number of scans and scan targets may be inspected. The main window of WVS can be seen in Figure 3.5.

![Acunetix Web Vulnerability Scanner](image)

**Figure 3.5** Acunetix Web Vulnerability Scanner.
Results of a scanning session are stored in a database by WVS. These results can be reported in many formats such as the Rich Text Format (RTF), Portable Document Format (PDF), or as HTML files. The reports can be generated with reporting templates. These templates allow customizing the level of detail of reports.

WVS can be automated by executing the `wvs_console.exe` binary from the Windows command line. The automation capabilities include running a new scan and reporting scanning results. However, we found the command line interface to be hazy for our purposes in VWS 5.1: a trial and error approach was required to clarify the exact behavior of the interface. Also, scanning was restricted for one scanning thread per time, so only a graphical user interface or command line interface scan could be run at the same time.

### 3.4 Summary

Software integration is the practice of assembling a set of software components/subsystems to produce a single, unified software system that supports some need of an organization [Stavridou, 1999, 3]. Continuous integration (CI) as a practice means that software is integrated with a certain schedule. A typical CI process performs an integration when the developers commit code to a version control system. This, with other CI practices, such as continuous testing, can benefit organizations and ease the integration efforts [Miller, 2008; Cannizzo et al., 2008].

The popularity of web applications has made them an attractive target for attacks. For example, online banking applications are an interesting target for attackers. Currently, cross-site scripting (XSS) and SQL injection vulnerabilities are the most widely spread web application vulnerabilities [Lucca et al., 2004; Kals et al., 2006; CVE, 2009]. To uncover these and other types of security vulnerabilities, security testing tools can be used [Curphey et al., 2006].

In the next chapter, a software process improvement technique for continuous integration is discussed. The technique forms a systematic testing approach for uncovering various types of web application security vulnerabilities.
4 THE PROPOSED SOFTWARE PROCESS IMPROVEMENT

4.1 Background

4.1.1 Motivation

Ambientia [Ambientia, 2009] is a company specialized in web application development. Most of the projects in the company are implemented with the Java Enterprise Edition [Allamaraju et al., 2000]. The company had experiments with continuous integration starting in 2004 with CruiseControl [CruiseControl, 2009]. Beginning in 2006, the prereleases of Atlassian Bamboo [Atlassian, 2009] were tested. As Bamboo 1.0 was released in 2007, it was chosen as the continuous integration server for the company. Bamboo was chosen because of its stability and ease of use compared to other available CI servers. The project teams were encouraged to use Bamboo. In some projects JUnit [Massol and Husted, 2003] was used to execute unit tests in conjunction with Bamboo. This provided additional test feedback.

The company emphasizes lasting customer relationships and reliability as a vendor. Lasting customer relationships can be maintained if a vendor can prove to be dependable. Late project deliveries or deliveries that do not match customers’ expectations can damage customer relationships. Therefore, the company wanted to develop a technique for improving software quality. As the eased integration efforts proved continuous integration beneficial to the company, it was thought that expanding the continuous integration activities was a good idea. The project for implementing a continuous integration process for automated testing activities started in 2007.

The strategical reasons for starting the project were driven by two observations. Firstly, it was felt that the increased competition in the market demanded measures which can provide the company competitive advantage. The technique was not expected to increase profits but, rather, increase company’s reliability as a vendor. Secondly, having a technique that other competitors in the market do not have can differentiate the company from other competitors.

As an objective, automated web application security testing was chosen. The testing objective can be argued. However, we felt that the market strongly demanded security testing. In fact, some of the customers require web application security testing as a standard procedure. Ultimately, we wanted the process to be flexible to include other types of testing (such as performance testing) as well. In
the end, we set three goals for the project. The company’s software development process should be improved by targeting these goals:

- The continuous integration and testing process will be standardized and established as a practice.
- Technical problems of web applications should be discovered in the early stages of development and continuously during a project.
- Developers can get continuous feedback on the state of the project.

4.1.2 Business requirements

We had positive experiences with Atlassian Bamboo so it was required to be the CI server. It was also though that, converting our projects to a different CI server would have probably been challenging. However, a tool for web application security testing was needed. The business requirements for the security testing tool are listed below:

**Automation capabilities** were required for including the tool in the CI process. Many of the tools evaluated did not provide any programmable interface for test automation.

**Fixed costs.** Because of potentially needing to test unlimited amount of builds we needed a tool with fixed costs and no restrictions on what we could test and in what scope. This ruled out some tools, software as service (SaaS) based tools. Some of the tools were limited in that sense that they allowed the scanning of only a limited number of web applications (such as web applications on one server).

**Customer and developer friendly reporting.** The tool needed to provide different kind of reports: for customers a high level overview of security of the web application and for developers a detailed report for fixing the vulnerabilities. This requirement implied flexible reporting with different reporting templates.

**Black box scanner.** We wanted to emphasize the "hacker’s view" of security testing. This excluded white box testing tools from our tool selection. However, the CI process could contain white box testing but black box scanning is the minimum requirement.
Automatic test case generation. We wanted to limit the effort required to begin testing with the tool. Thus, the tool was required to generate test cases automatically. Most of the tools also provided a feature to manually capture the test cases besides automatic test case generation. In most cases, automatic test case generation meant that the tool was able to crawl the structure of the web application automatically and execute the test cases to the web pages discovered.

After evaluating various security testing tools we chose Acunetix Web Vulnerability Scanner (Section 3.3.4). It is necessary to notice that the evaluation was based on our situation and a security testing tool would probably be very different with different business requirements.

4.1.3 Company’s project model

In Figure 4.1 is depicted the software process which is utilized by the company. Stage-wise, the model is sequential and has six stages beginning from the initial system requirements, and ending as a deployed web application. From this point of view, the model is comparable to Royce’s model where the process is supposed to progress in linear fashion. However, the model highly emphasizes risk-driven, iterative approach to software development comparable to Boehm’s spiral model. Furthermore, the model emphasizes close cooperation between the customer and the development team. In the first stage, consulting, the system requirements are not assumed to be known well. Therefore, the concept design stage is the stage, where most of the requirements should be elaborated with two approaches. Firstly, a (visual) prototyping approach is used for discovering requirements problems. Secondly, system requirements are derived from the prototype. The next stage, elaboration, should address more technical nature of matters such as development of system architecture and the user interface.

Naturally, one would ask how the iterative approach of this project model is defined. Here, it is important to emphasize the risk-driven nature of the model again. The concept design, elaboration, construction and release testing stages are iterative. As an example, lets say there is a project with three iterations in the concept design stage. This would mean that there would be three software prototypes produced before the elaboration stage, each prototype further clarifying the requirements. In this way, the customer is able to see something concrete as soon as possible. The number of iterations is not always \textit{a priori} defined but rather the decision of the project organization in charge. In this sense,
the process is adaptive. Of course, the project schedules most often face fixed deadlines. After the elaboration and construction stages are completed, the result should be a web application which can be tested with all features defined by the customer. The deployment stage is the combination of release ceremonies and further maintenance. The maintenance tasks are often performed with a light weight process.

This far, we have been describing the process of this model. A question which should be addressed is the methodological aspects of the model. For each stage, the best practices and methods are defined. As a good practice the model could say, for example, that in the construction stage, continuous integration should be used. The methodology improvement group meets weekly to address the problems discovered in the projects. The function of the group is to provide solutions to everyday development problems and also to improve the existing methodology based on the feedback. For example, the methodology improvement group could
improve the practice of unit testing based on the current project experiences. From each team, at least one team member should be present in the methodology improvement meetings. In this way, the feedback from the whole organization can be aggregated.

4.2 Software process improvement

The software process improvement enforces automated functional testing activities. In Figure 4.2 is an overall depiction of the process, which includes four software process improvements: (I) executing commit builds, (II) executing functional builds, (III) executing security scanning, and (IV) publishing vulnerability reports.

![Figure 4.2 Continuous build testing](image)

**Figure 4.2** Continuous build testing. Firstly, a commit build is executed. This provides immediate feedback for the project members. A successful commit build triggers a functional build. The functional build is tested with system tests: in this case, the system tests consist of security testing. After security scanning has been executed, the vulnerabilities uncovered are published as vulnerability reports. These vulnerability reports allow project members to get feedback for the next commit build. For example, a discovered vulnerability might be fixed in a new build.
4.2.1 SPI I: executing commit builds

**Distributed building with Bamboo**

The primary function of a commit build is to provide immediate feedback for the project members. We introduced a typical continuous integration process in Section 3.2. The simplified CI process introduced in Figure 3.2 does not scale if the number of builds increases. To overcome the limitations of a single build environment Atlassian Bamboo introduced distributed builds in the version 2.0 [Atlassian, 2009]. Instead of a single CI server, the builds can be distributed among build nodes. The arrangement can be seen in Figure 4.3.

As a prerequisite, all projects are on a version control system, namely Subversion. When a developer makes a commit to the version control system, Bamboo notices the commit. Processing the commits are distributed to a build cluster with multiple nodes. Any of the nodes attached to the build cluster can be triggered for a build. This approach can decrease build times: if one node is busy, another (idle) build cluster node can handle the build. Also, as a fail-safe mechanism, multiple build nodes increase redundancy, and thus, increase reliability.

Bamboo implements distributed builds with a master build server. Attached to the master build server is one or many build nodes. These build nodes execute the Bamboo Remote Agent. The remote agents are responsible for handling the builds on the build nodes and communicating with the master build server. We noticed some problems with the fault tolerance of remote agents in the version 2.0. If the master build server went down the agents were unable to recover connections to the master even if the master server came back online.

---

**Figure 4.3** Distributed builds with Atlassian Bamboo.
Despite the problems with the fault tolerance issue our experiences showed that distributed building helped to increase throughput from our build environment. During the initial implementation we moved from a single build machine to three build nodes. We did not experience difficulties in migrating our environment from Bamboo 1.0 to 2.0.

Publishing build artifacts

The continuous testing process includes system testing. Therefore, it is necessary to store the build artifacts for later execution on a web application server. For this purpose, a web application archive (WAR) file [Allamaraju et al., 2000, 452] is created. The WAR files can be deployed on an application server that is, the application server reads the WAR file and prepares the web application for execution. After the WAR file is created, the file is copied to a package repository. The package repository is a web server. The WAR files can be retrieved from the package repository with a HTTP request such as http://pkg.example.com/build/prj-plan-14.war

Atlassian Bamboo identifies builds with a build key (i.e. PRJ-PLAN) and a build number (i.e. 14). These properties are very helpful for identifying a build later on. For example, we could refer to a build as PRJ-PLAN-14. The WAR files have the same build naming convention. The previous example could be named as the WAR file prj-plan-14.war. To expose these build properties to the build scripts, one needs to specify them as a builder properties in Bamboo (-DbuildKey=${bamboo.buildKey} -DbuildNumber=${bamboo.buildNumber}).

```
<project>
  <property name="deploy-tasks-remote*
       value="/opt/bamboo-scripts/deploy-tasks-remote.xml" />
  <target name="war">
    <property name="warPath" value="/tmp" />
    <property name="warFile" value="${buildKey}-${buildNumber}.war" />
    <echo message="Creating WAR package ${warFile}..." />
    <war destfile="${warPath}/${warFile}" webxml="/www/example/WEB-INF/web.xml" basedir="/www/example" includes="**" excludes="WEB-INF/logs/*.*.log">  
  </war>
</target>
<target name="deploy-remote-build*" depends="war">  
  <ant antfile="${deploy-tasks-remote}" target="war-deploy" />
</target>
</project>
```


The publishing process has two stages. First, the web application is packaged
as a WAR file. This can be done with Ant’s WAR task (Listing 4.1). Next, the
build scripts calls the script deploy-tasks-remote.xml (Listing 4.2) to copy the
WAR file to the package repository. The script uses the SCP (secure copy)
protocol for file transfer. After executing the script, the transferred WAR file is
available from the repository with the HTTP protocol.

```xml
<project name="deploy-remote">
  <target name="war-deploy">
    <property name="deployRemoteUser" value="war-deploy" />
    <property name="deployRemoteHost" value="pkg.example.com" />
    <property name="deployRemoteDir" value="/www/build" />
    <property name="sshKeyFile" value="/home/deploy/.ssh/id_rsa" />
    <echo>Copying ${warFile} to ${deployRemoteHost}...</echo>
    <scp trust="true" localFile="${warPath}/${warFile}" keyfile="${sshKeyFile}" passphrase=""
todir="${deployRemoteUser}@${deployRemoteHost}:${deployRemoteDir}/${warFileLowerCase}" />
  </target>
</project>
```

Listing 4.2: File: deploy-tasks-remote.xml (abbreviated). Copies a WAR file into
the remote package repository.

The deployment process worked reliably. From our experiences, depending on
the web application (WAR file) size, packaging and transferring a build consumes
some time but it is not a significant overhead for commit builds. The problematic
part is that publishing WAR files consumes space. Say, you have 50 commits per
day and the size of each WAR file is 50 megabytes. This consumes 2.5 gigabytes
of space daily. Over a month, roughly 50GB of space is required. We found out
the need to clean the old builds. We developed a process which deletes any WAR
file older than seven days.

4.2.2 SPI II: executing functional builds

A functional build can be only executed if the preceding commit build succeeded.
For example, if the commit build failed to compile source code, the same failure
would cause the functional build to fail, too. Naturally, a functional build would
fail even if the commit build was successful in source code compiling. Functional
builds can be triggered with Bamboo’s build plan dependencies. In Bamboo, if
a plan is dependent on another plan, it is called a child plan. When the non-
dependent plan (parent plan) successfully finishes, the child plan is triggered. In
our implementation the functional build has two tasks: (1) deploying the web
application, and (2) executing functional tests.
Deploying web application

Now, assume that the web application is available as a WAR file. For executing system tests, it is necessary to deploy the WAR files created on an application server. A temporary application server is created matching each build. Deployment is done using a WAR file pulled from the package repository.

The task enqueue-security-test in security-test.xml executes a functional build. Firstly, because the application server is created dynamically, a free server port must be found. For this purpose the code in Listing 4.3 is executed.

```groovy
<groovy><![CDATA[
    def port = 16280
    while (port < 60000) {
        try {
            s = new Socket("localhost", port)
            // No exception is thrown if socket is in use
        } catch (Exception ex) {
            // Socket could not be connected to -> it’s free
            properties['http.port'] = port
            break
        }
        port += 100
    }]]>
<echo message="Using port ${ http.port } for Tomcat" />
```

Listing 4.3: File: security-test.xml (abbreviated). Finds a free port for the application server.

After a server port is found, the deployment URL is known for the build. For example, we could deploy the build to the URL http://build.example.com:16284. The next step is to actually create the application server (Listing 4.4). Deployment is done with the previously created WAR file.

```groovy
<exec executable="/www/scripts/launch_tomcat.sh">  
    <arg value="${ warFile }" />
    <arg value="${ http.port }" />
</exec>
```

Listing 4.4: File: security-test.xml (abbreviated). Launches the application server.

The application server is created with the script launch_tomcat.sh. The script gets the WAR file to be deployed from the package repository with a HTTP request. After that, the war file is unpacked to a deployment directory. Then
the application server configuration (server.xml) is generated. The server port and the deployment directory need to be specified in the server configuration. The last action is to launch the application server. As a result, the build can be accessed from the deployment URL. This allows functional tests to be executed. As a sanity check, as the last step, it is verified that the deployment was actually successful and the server responds (Listing 4.5).

```xml
<target name="test-http-host">
  <description/>
  <property name="timeoutSeconds" value="60"/>
  <echo message="Waiting for ${appUrl} to respond..."/>
  <waitfor maxwait="${timeoutSeconds}" maxwaitunit="second"
    timeoutproperty="timedOut">
    <and>
      <socket server="${http.host}" port="${http.port}"/>
      <http url="${appUrl}"/>
    </and>
  </waitfor>
  <fail if="timedOut" message="Failed to detect application running in ${timeoutSeconds} seconds"/>
  <echo message="Application found running at ${appUrl}!"/>
</target>
```


Again, we found resource issues to be problematic. The application servers consume significant amount of memory. We assumed that a temporary application server should be generated for each build waiting for test execution. We realized that a better solution would be a finite number of predefined application servers for testing. As shown in Section 4.2.3 no simultaneous builds can be scanned. Therefore, we might as well deploy the build to be tested prior scanning. Also, the complexity of deploying to application servers may be better tackled with some existing tools such as Cargo [Cargo, 2009]. However, the implementation of deployment mechanism does not affect the security scanning process.

**Executing functional tests**

For the reasons described in Section 4.2.3 the tests cannot immediately be executed. Therefore, as the build is deployed and while waiting for testing, a request to execute the functional tests is sent. This is done with the script `add_scan_target.groovy` (Listing 4.6).

4.2.3 SPI III: executing security scanning

Checking for queued builds

As described before, Acunetix Web Vulnerability Scanner supports only one scanning session per time. Our conclusion is that there is no technical reason for the limitation, but the limitation is rather introduced by software licensing reasons. This was one of the reasons why we developed a queuing system for security scans. The other reason was that we wanted to have our own well-defined interface for controlling automation. If there are changes in WVS, we can hopefully change our implementation of interface rather than changing the build process itself.

The client library we developed for Acunetix Web Vulnerability Scanner is called `acunetix-console-client`. The client is a stand-alone Java application meant to be run as a background server process. The client abstracts the interaction with the command-line interface (WVS_CONSOLE.EXE) provided by WVS. The main functionality of the client is to maintain the scan queue. As we mentioned in Section 4.2.2, the functional builds send scan requests to the client. A scan request contains the URL of the web application build to be security scanned. At any time, WVS may be busy scanning a web application. This means that the remaining scan requests have to wait in the scan queue.

In Figure 4.4 is described the architecture of acunetix-console-client. WVS is not actually designed to run as a server application but rather a stand-alone Microsoft Windows application used by a tester. To overcome this limitation, WVS was installed on a virtualized Windows XP workstation executing on the Vmware Server [Vmware, 2009]. In Figure, Vmware is described as "security testing server". Because the rest of our build environment runs on Linux, we wanted to have minimal dependencies on the Windows workstation. Therefore, we designed acunetix-console-client to be run on the Linux-based build nodes. In Figure, acunetix-console-client is running on one of the build nodes.
Figure 4.4 Implementation of acunetix-console-client.

Because acunetix-console-client and WVS are separated to servers, the two processes need to exchange messages. For example, the client needs to launch a new security scan. SSH protocol is capable of executing commands remotely so it was chosen as the interface. In Figure 4.4 the communication between the client and WVS can be seen in the arrow ”execute wvs_console.exe with SSH”. SSH Tectia Server [SSH, 2009] was installed on the Windows XP workstation. The remote command execution was implemented with the Trilead SSH for Java library [Trilead, 2009].

The scan command which is executed by acunetix-console-client can be seen in Listing 4.7. It is important to notice that the scan results are stored in a database for later retrieval. Also, an XML file containing the results is generated.

Listing 4.7: Launching a security scan with wvs_console.exe.

```
"C:\Program files\Acunetix\Web Vulnerability Scanner 5\wvs_console.exe"
/Scan http://build.example.com:16284 /ExportXML C:\temp\results.xml
/Save C:\temp\results.wvs /SavetoDatabase
```

Automatic test case generation

Automatic test case generation was one of the business requirements for the security testing tool (Section 4.1.2). In Section 2.4.4 code-based, interface-based and specification-based approaches for automating test case design were introduced. In our implementation, interface-based approach was used for test generation.

Approaches exist for automating test case generation for web applications in the literature [Benedikt et al., 2002; Miao et al., 2008]. Probably the most common approach is to systematically discover the web application structure. This approach is called web application crawling. Benedikt et al. describe some common challenges and design principles related to web application crawling in
the implementation of VeriWeb, an automated testing tool for web applications [Benedikt et al., 2002]. With dynamic web applications, the challenges include form submissions and execution of client-side scripts [Benedikt et al., 2002, 2].

In our implementation crawling is delegated to WVS. As previously described, acunetix-console-client retrieves a build from the scan queue for testing. The crawler of WVS is initialized with the test build as the starting URL. WVS starts systematically browsing the test build from the start URL. The exact crawling algorithm of WVS is not known to us. It is fair to assume that the crawler is based on the breadth-first search (BFS) [Cormen et al., 2001, 531] because parameters such as crawling depth can be set. When the crawling is finished, WVS returns a set URLs for executing tests.

In practice crawling seemed to be somewhat problematic. For a web application with a small number views (say ten distinct URLs) crawling can be finished in a short time. When the number of views (web pages) in a web application increases also the crawling time increases. One could simply assume that the problem could be solved by limiting the crawling depth or the number of pages to be crawled. This, however, is not a good approach since a vulnerable page can be found anywhere in the web application. The problem is related to test automation in general. It is hard to design good test cases automatically because the crawler cannot prioritize the different views in the same way as a human tester.

Acunetix Web Vulnerability Scanner tries to ease the problem in the latest version with combining some white box testing practices with black box scanning. The AcuSensor component is placed on the web server and can report all the files present and accessible though the web server to the scanner component. We did not have a chance to test this approach but it would be interesting as a future work.

**Executing test cases**

As described in the previous section, the crawler component of WVS uncovers the structure of the web application. The crawler returns a set of URLs for which the test cases are executed. Lets say there is the following set of URLs: {http://www.example.com/news?id=1, http://www.example.com/feedback_form, ...}. A number of test cases is executed for each of these URLs. WVS performs modified HTTP requests to test vulnerabilities such as SQL injections and cross-site scripting vulnerabilities. For example, for the first URL, an apostrophe could be added to the URL. A test request could be performed with the address http:
If the web application displays signs of errors in its output, the test is reported as failed. In addition to these tests, WVS tries to exploit commonly known vulnerable URLs, such as the presence of unnecessary administration interfaces.

What is notable is the quadratic \( (n^2) \) running time [Cormen et al., 2001, 25] of the testing algorithm. Let's say the size of the set of crawled URLs is \( u \). The number of tests to be executed for each URL is \( t \). This implies that the minimum number of executed tests (HTTP requests) is \( ut \), thus the running time is \( n^2 \). Web applications with a lot of URLs to be tested are problematic. Because of the quadratic running time, the testing times tend to increase. Let's make an assumption that the time for executing a single test is one second. The number of tests to be executed for each URL is 100. With these assumptions, a web application with 10 distinct URLs can be tested with \( 10 \times 100 = 1000 \) requests (17 minutes) but a web application with 5000 URLs takes \( 5000 \times 100 = 500000 \) requests (139 hours) to scan!

4.2.4 SPI IV: publishing vulnerability reports

So, this far, we have described the processes for executing commit and functional builds, and, last, the execution of security scanning. As the last process continuous testing includes vulnerability report publishing. The purpose of vulnerability report publishing is to provide feedback for the project members of the vulnerabilities found in the security scanning. The feedback is provided in the form of vulnerability reports. The reports include test execution results such as SQL injection and cross-site scripting vulnerabilities uncovered. The vulnerability reports are published on a web site for later retrieval. The two stages of publishing the reports are covered in the following subsections.

Report generation

As previously noted, the results of a security scan are stored in an XML file and a database. The database containing the vulnerability results is an MDB database file (`Data\Database\vulnscanresults.mdb`) written with the Microsoft Jet Database Engine [Haught and Ferguson, 1997]. WVS provides the `REPORTER_CONSOLE.EXE` command line tool for generating reports from the database. Unfortunately we found the command line interface of `reporter_console.exe` to be tricky. Report generation with `reporter_console` requires a "scan ID" which is assigned by WVS. Unfortunately the reporting tool provides no way of retrieving
the scan ID for a specific scan session.

To overcome the limitations with reporter_conso le, we created a tool. The tool, acunetix-scan-reporter, retrieves the corresponding scan session from the database and executes report generation with reporter_conso le.exe. For example, if the build was scanned from the URL http://build.example.com:16284, the tool tries to retrieve the scan ID from the first row (latest result) matching the URL. For reading the database, Jackcess [Jackcess, 2009] was used. The command executed by acunetix-scan-reporter can be seen in Listing 4.8.

```
"C:\Program Files\Acunetix\Web Vulnerability Scanner 5\reporter_conso le.exe"
/Report WVSINGLESCAN.rep /Target <scan ID from the database>
/Action HTML /Output C:\Temp\report.html
```

Listing 4.8: Generating an HTML report with reporter_conso le.exe.

**Publishing**

There is little or no use for the scan reports if they are not published. Therefore, after a report is generated, it is published to a web site. Of course, as a security measure, the web site remains as an intranet web site and no access is granted outside the company network.

![Figure 4.5 The scan queue view.](image)

The web site can be seen in Figure 4.5. Firstly, the user can see the scan queue (last scans). For those scans which are completed the scan report can be
seen. If the scan uncovered vulnerabilities from the scanned web application, an error status is indicated. Otherwise, the scan is reported as successful. By clicking an individual scan report (the document icon) the scan report opens. A sample scan report can be seen in Figure 4.6. We noticed that it might be a useful feature to allow to add web sites to be scanned to the scan queue manually (the scan button, Figure 4.5). In fact, there is no limitation on what a user can scan manually.

![Sample results of a scan.](image)

Publishing is implemented in the acunetix-console-client. Firstly, the error count is parsed from the XML file containing scan results. This is done by counting the `ReportItem` elements. Secondly, the HTML report is transferred with the SCP (secure copy) protocol to the web server.

From our point of view the reporting process provides tremendous value. Conventionally the test results are stored in a document (such as PDF or RTF) and distributed to project members with some means (such as e-mail). The ability to view the test results with a web browser reduces the time to follow the test results and makes easy to share the test results within a project team. Currently our implementation lacks the ability to use different reporting templates so the reports are made mostly for persons with technical knowledge. Also, we do not have experience of how efficient the reports are displaying the vulnerability infor-
mation, if, for instance, a project produces a lot of builds. That is, comparing the vulnerabilities between different build versions is a need that must be addressed because it will reveal significant information and feedback.

4.3 Summary

This chapter presented a continuous integration process for automated testing activities. The process provides rapid and automatic feedback on the security of the web applications under development. Four software process improvements were proposed.

- **Commit builds** provide immediate feedback for the project team of compilation failures. Also, a commit build should store the build artifacts to a package repository. For efficiency, the commit builds are handled in distributed manner with a build cluster with the ability for simultaneous processing.

- The "testing weight" tends to increase when moving from unit to integration and functional to acceptance testing. Therefore, a functional build is produced for executing system tests and/or acceptance tests.

- Functional builds are tested with **security scanning**. The process inspects the web application by crawling the web application structure and executing tests for uncovering security vulnerabilities.

- The **vulnerability reports** are published for project stakeholders. The reports contain the vulnerabilities detected based on the test execution.

In the next chapter, remarks from the implementation of the process are made. We also propose further improvements for the process.
5 Conclusions

5.1 Project retrospect

The nature of the software process improvement was to concentrate on the software engineering part of the process. What resulted is that we (1) evaluated the supporting tools for the implementation (Section 4.1.2), and (2) proposed four software process improvements (Sections 4.2.1, 4.2.2, 4.2.3, 4.2.4). Some remarks can be made from the project.

The testing objective of the project was challenging: we designed a process where all commit builds are tested with functional tests. As stated in Section 3.2.1 the solutions from industry do not support this approach. For example, Cannizzo et al. describe an approach where performance and robustness builds are executed during weekend time [Cannizzo et al., 2008]. The problem is that the “testing weight” tends to increase when moving from unit to integration and functional to acceptance testing [Duvall et al., 2007, 143]. We would suggest careful planning about the testing strategy for similar projects: commit builds should provide immediate feedback and the most heavyweight automated testing should happen during night time or weekends.

Test automation. Our experiences let us to believe that the security testing tool vendors were not prepared for supporting test automation. Web application security testing is seen as an activity which is often performed by a tester manually. This made the implementation more challenging: if a tool does not provide an established approach for test automation, it is hard to add the support later on. We hope that the tool vendors would add further support for test automation.

Start small. Organization must be ready for automation. We found out that projects had many different quality standards and practices. In fact, not all projects could execute commit builds. We would say that an organization must be able to achieve a base-line in quality and automation before advanced testing approaches can be fully utilized. Say, 90% of the projects should be execute commit builds before functional builds can be utilized. This implies that the change towards test automation takes time and management effort. However, we found out that some of our work was not directly beneficial to continuous testing but instead benefited the organization as a whole. For example, the distributed
build system helped the organization to provide commit builds in a more efficient manner (Section 4.2.1).

**Management support.** In the long run, test automation cannot succeed if there is no absolute commitment and careful strategical planning from the management. Fewster and Graham describe the problem well: "Test automation is an infrastructure issue, not just a project issue. In larger organizations, test automation can be rarely justified on the basis of a single project, since the project will bear all of the start-up costs and teething problems and may reap little of benefits. If the scope of test automation is only for one project, people will then be assigned to new projects, and the impetus will be lost" [Fewster and Graham, 1999, 12].

**Organizational impact**

In this short time, we were not able to introduce fundamental change to the testing habits of the company but rather provided empirical evidence that indeed, this kind of testing is possible and implementable. This, itself, is a valuable result for the company. From our point of view, the organizational lessons learnt justify the project. In addition, Acunetix Web Vulnerability Scanner is now used for security testing. Most of the testing with the tool is currently manual (executed by a tester) but can still provide extremely useful feedback for the developers. Because of introducing systematic security testing the developers are more aware of web application security risks. As we also noted, the project resulted in improvements of the basic infrastructure. These improvements are beneficial for the whole organization.

**5.2 Future work**

We have evidence that builds can be security tested with the process we engineered. In Chapter 4 we provided some remarks on how the process could be improved. We believe that with further development (such as having nightly builds) the current process could provide significant value for an organization. We also believe that continuous integration and automated functional tests will be more widely utilized in the future. Our hope is that automated functional tests are used as a de facto practice in the same manner as unit tests today.
REFERENCES


