Testing in Scrum

A Guide for Software Quality Assurance in the Agile World

Tilo Linz

August 2014

EuroSTAR
Software Testing Conference

Is This Book Right For Me?

INTRODUCTORY

Introductory content is for software testing professionals who are relatively new to the subject matter of the ebook. Introductory ebooks typically feature an overview of an aspect of testing or guidance on understanding the fundamentals of the subject at hand.

INTERMEDIATE

Intermediate ebooks are for software testers who are already familiar with the subject matter of the eBook but have limited hands-on experience in this area. Intermediate level usually focuses on working examples of a concept or relaying experiences of applying the approach in a particular context.

ADVANCED

Advanced ebook content is for software testers who are, or intend to become, experts on the subject matter. These ebooks look at more advanced features of an aspect of software testing and help the reader to develop a more complete understanding of the subject.

Abstract

These days, the software development for more and more projects is being carried out using agile methods, such as Scrum, XP or other agile methods. Even in large-scale projects and product development units, traditional approaches, such as development according to V-models, have been rejected in favour of agile approaches like Scrum.

The book – from a project- and test-management perspective – explains and discuss, how software testing and quality assurance works in agile environments. This is explained in the context of Scrum and also illustrated by several case studies.

Doing this the book covers the new ISTQB® Syllabus for 'agile Software Testing' and thus will be a relevant source for all students and trainees worldwide who will undertake this new ISTQB certification (currently more than 250 000 people worldwide have passed the certification for the basis Syllabus 'ISTQB Foundation Level'- on which the 'agile Software Testing' is build upon).

The material provided here as free ebook is taken from chapters:

- 5 - Integration Testing and Continuous Integration
- 6 - System Testing and Testing Nonstop
- and case study 8.2 - Nonstop System Testing

1st edition 2014 Copyright © Tilo Linz

Published by Rock Nook, Inc.
www.rockynook.com

All rights reserved. No part of the material protected by this copyright notice may be reproduced or utilized in any form, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system, without written permission of the publisher.

Biography

Tilo Linz has been working in the software quality assurance and testing field for more than 20 years now. He is one of the founders and chairman of imbus AG – one of the leading german service, product and training providers in this sector.

Beside imbus Tilo Linz' commitment particularly focuses on the vocational education and training opportunities for test experts. He helped to initiate the International Testing Qualifications Board (ISTQB) and launched it as first chairman (2002-2005). In 2002 he also founded the German Testing Board e.V. (GBT), where he was chairman on a voluntary basis till 2013.

Not least through his publications as (co-)author of the books "Software Testing Foundations", "Software Testing Practice: Test Management" and "Testing in Scrum" he is ranked among the most renowned experts in the field of software testing.

Table of Contents

1. Acknowledgments

Even if it is my name that appears on the cover, this book wouldn't have been possible without the advice and support of many of my colleagues.
I would particularly like to thank the interviewees and authors of the case studies, who also acted as reviewers and sounding boards: Dr. Stephan Albrecht at Avid, Dierk Engelhardt at imbus, Andrea Heck at Siemens, Eric Hentschel at ImmobilienScout24, Sabine Herrmann at zooplus, Joachim Hofer at imbus and Terry Zuo at GE Oil & Gas. The many interesting conversations we had provided plenty of valuable insights into the implementation of agile development techniques and the real-world use of Scrum.

I would also like to thank my expert reviewers for their valuable comments, suggestions, and corrections: Oliver Gradl at Siemens for his input on agile integration and system testing, Dr. Stefan Kriebel at BMW and Horst Pohlmann for their feedback on embedded systems, Stefan Schmitz at iq-stz for his profound knowledge of the ISO 9000 standard and auditing, Uwe Vigenschow at oose Innovative Informatik for the fruitful discussions on the subject of acceptance testing and Prof. Mario Winter at the University of Cologne who, in spite of being involved in a book project of his own, contributed his time as a reviewer and offered important input for the chapter on integration testing. Thanks also go to all the other reviewers whose names are not mentioned here.

Thanks, too, go to the entire staff at imbus for their valuable advice and time spent supporting this project, especially Arne Becher, Dr. Christian Brandes, Thomas Roßner and Carola Wittigschlager. My heartfelt thanks also go to Claudia Wissner, without whom many of the illustrations would not have progressed beyond the sketch stage.

Thanks also go out to Matthias Rossmanith at Rocky Nook for his tireless help during the production of this book and his patience when an extra sprint or two were required.

And, last but not least, a big thank you to my wife, Sabine, and our daughters, Lisa and Lena, who had to survive without me for many long evenings and weekends while I worked on the text. I wish you an interesting read and every success applying these techniques in your own testing environment.

- Tilo Linz, February 2014

Software is everywhere. Virtually every complex product manufactured today is controlled by software, and many commercial services are based on software systems. The quality of the software used is therefore a crucial factor when it comes to remaining competitive. The faster a company can integrate new software in its products or bring software products to market, the greater the opportunity to beat the competition.

Agile software development methods are designed to reduce time to market (TTM) and improve software quality while increasing the relevance of products to customer needs. It is no surprise that the use of agile methodology is on the increase in large international projects as well as in product development at all manner of large corporations. In most cases, this means switching from the tried and trusted V model to agile testing using Scrum methodology.

However, switching to agile and learning to use it effectively and productively is not always easy, especially when multiple teams are involved. Every team member, project manager and all members of line management have to make significant changes to the way they work. Particularly, the effectiveness of software testing and quality assurance (QA) are crucial to the success of introducing and using agile methodology in a team, a department or an entire company, and will determine whether its potential advantages can be put into practice.

Most literature on the subject of agile methodology (including many of the works referenced in the bibliography at the end of this book) is written from the viewpoint of software developers and programmers, and tends to place its main emphasis on programming techniques and agile project management—testing is usually only mentioned in the guise of unit testing and its associated tools. In other words, it concentrates on testing from the developer's point of view. However, unit tests alone are not sufficient and broader-based testing is crucial to the success of agile development processes.

The aim of this book is to close this gap and describe the agile development process from the viewpoint of testing and QA. It shows how agile testing works and details where traditional testing techniques are still necessary within the agile environment and how to embed these in the agile approach.

1.1 TARGET AUDIENCE

Understanding how testing in agile projects works:
On one hand, this book is aimed at beginners who are just starting to work with agile methodology, who are due to work in agile projects or who are planning to introduce (or have already introduced) Scrum into a project or team.

- We provide tips and advice on how product development managers, project managers, test managers, and QA managers can help to realize the full potential of agile methodology.

- Professional (certified) testers and software quality experts will learn how to work effectively in agile teams and make optimal use of their expertise. You will also learn how to adapt your working style to fit into the agile environment.

Augmenting your knowledge of (automated) testing and agile quality assurance techniques:
On the other hand, we are also aiming at readers who already have experience working in agile teams and wish to expand their knowledge of testing and QA techniques to increase productivity and product quality.

- Product Owners, Scrum Masters, QA operatives, and members of management teams will learn how systematic, highly automated testing works and about the importance of providing continual, reliable and comprehensive feedback on the quality of the software being developed.

- Programmers, testers, and other agile team members will learn how to apply highly automated testing methods to unit, integration and system tests. The text includes a wealth of practical examples and practice questions, making it well suited as a textbook or for self-teaching.

1.2 BOOK CONTENTS

Chapter 2 provides a brief overview of the currently popular Scrum and Kanban methodologies and an overview of the most important agile management methods for managers looking to implement agile methods in their departments. These methods are compared with traditional methods to give you an idea of the changes that introducing agile methods involves. Readers who are already familiar with Scrum and Kanban methodology can skip this chapter.

Chapter 3 details the lightweight planning and control tools that Scrum uses in place of traditional project planning tools. Remember, working in an agile fashion doesn't mean working aimlessly! Chapter 3, too, is aimed primarily at readers who are making the switch to agile development. Nevertheless, the explanations and tips regarding the importance of planning tools for constructive quality assurance and error prevention will be useful to experienced readers too.

Chapter 4 covers unit testing and "test first" programming techniques. Topics include what unit tests can be expected to achieve and how they can be automated. This chapter covers the basics of unit testing techniques and tools and will help system testers, test specialists, and project team members with little or no unit testing experience to work more closely and effectively with programmers and unit testers. It also includes a wealth of useful tips that will help experienced programmers and testers to improve their testing techniques. We also explain the test first test-driven approach to programming and its importance within agile projects.

Chapter 5 covers integration testing and the concept of continuous integration testing. Integration testing covers test cases that even the most diligent

programmers miss when using unit tests. This chapter covers all the basics of software integration and integration testing, and introduces continuous integration techniques, explaining how to use them within a project.

Chapter 6 discusses system testing and the concept of nonstop testing. Based on system testing foundations, this chapter explains how to use agile techniques to perform manual tests. It also explains how to automate system tests and embed them in the continuous integration process. This chapter is aimed not only at system testers and other testing specialists, but also at programmers who wish to better understand agile testing techniques that lie outside their usual development-driven remit.

Chapter 7 compares traditional and agile ideas of quality assurance and explains the constructive and preventative QA practices that are built in to Scrum. This chapter includes tips on how to perform agile QA and explains how all QA specialists can use their know-how within agile projects and contribute to the success of agile projects.

Chapter 8 presents six industrial, e-commerce, and software development case studies. These reflect the experiences gained and lessons learned by the interviewees during the introduction and application of agile techniques.

Chapters 2, 3, 7, and 8 discuss process and management topics and are aimed at readers who are more interested in the management aspect of the subject. Chapters 4, 5, and 6 discuss (automated) agile testing at various levels and are oriented toward more technically minded readers. Because agile testing is not the same as unit, we also explain in detail the aims of and differences between unit, integration, and system testing.

Figure 1 provides a visual overview of the book's structure:

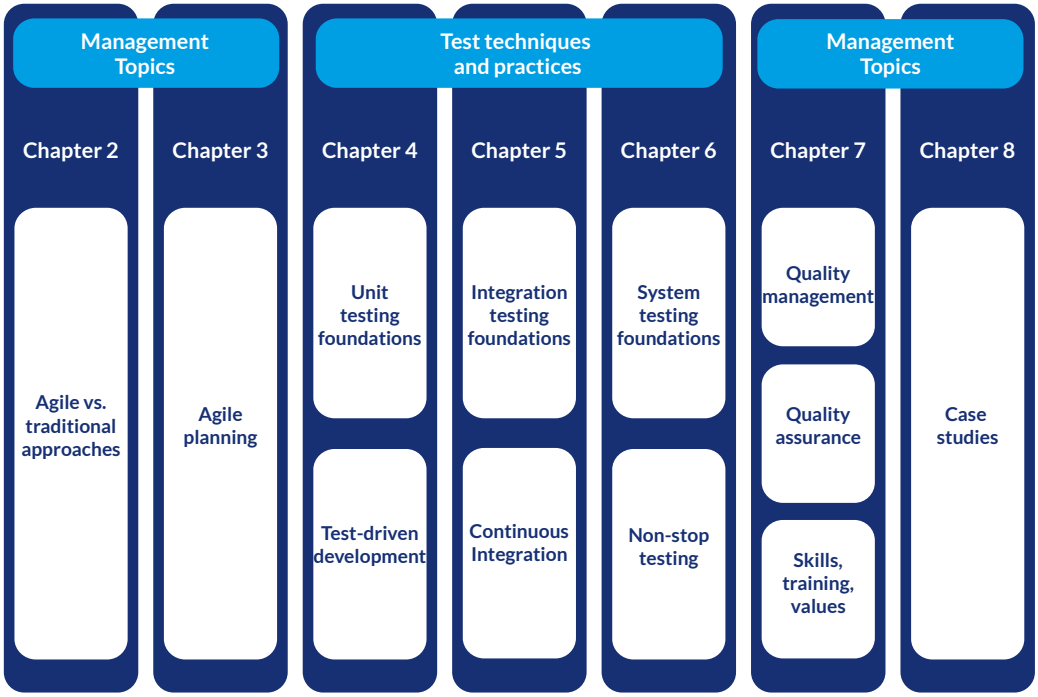

Figure 1. The structure of the book

Case study, checks and exercises:
According to most popular literature on the subject, many agile concepts, techniques and practices are simple to implement. Similarly, the ideas, advice and tips covered in the following chapters may at first appear simple, but the sticking points often only become evident in the course of implementation. To help you understand and experience the challenges involved, the text includes:

- A fictional case study that illustrates the methodology and techniques being discussed.
- Checks and exercises to help you recap the content discussed in each chapter and scrutinize your own situation and behavior within your project.

1.3 CASE STUDY

The fictional case study the book refers to uses the following scenario. The company eHome Tools is a developer of home automation systems based on the following elements:

- Actuators:
  Lamps and other electrical devices are connected by means of electronic switches. Every actuator is connected by wire or wirelessly to a comms bus that can be used to control it remotely.

- Sensors:
  Thermal, wind and humidity detectors and simple contact sensors (for example, to indicate an open window) can also be connected to the bus.

- Bus:
  Switching instructions and status signals for the actuators and sensors are sent via the bus to the central controller in the form of “telegrams.”

- Controller:
  The central controller sends switching signals to the actuators (e.g., “switch kitchen light on”) and receives status signals from the sensors (e.g., “kitchen temperature 20 degrees”) and actuators (e.g., “kitchen light is on”). The controller is either event-driven (i.e., reacts to incoming signals) or works on the basis of timed commands (e.g., “8pm -> close kitchen blind”).

- User interface:
  The controller has a user interface that visualizes the current status of all elements of the eHome and enables its inhabitants to send instructions (e.g., “switch kitchen light off ”) via mouse click to the house systems.

eHome Tools has many competitors and, in order to remain competitive, the company decides to develop new controller software. An increasing number of customers is asking for software that can be controlled via smartphones and other mobile devices, so it is clear from the start that this project has to be completed quickly if it is to be successful. It is essential that the new system is extensible and open for third-party devices. If the new system is able to control devices from other manufacturers, management is convinced that the company can win over customers who wish to extend their existing systems. To fulfill this goal, the new system has to support as wide a range of existing third-party hardware as possible and must be capable of adapting to support new devices as soon as they appear on the market.

In view of these challenges, the decision is taken to develop the new system using agile methodology and to release an updated version of the controller software (with support for new devices and protocols) every month.

1.4 WEBSITE

The code samples included in the text are available for download at the book’s website [URL: SWT-knowledge]. Feel free to use them to construct your own test cases.

The practice questions and exercises are also available online and can be commented on. We look forward to discussing your comments and suggestions.

In spite of all our efforts and those of the publisher, it is still possible that some text errors remain. Any necessary corrections will be published online.

This chapter explains the differences between unit and integration testing, how to design integration test cases and how to embed them in a fully automated Continuous Integration (CI) environment along with the unit tests we have already written.

3.1 INTEGRATION TESTING

A software system is made up of multiple components. For the system to work properly, each component has to function reliably on its own and, most importantly, all the components have to work together as planned. Integration testing checks whether this is the case.

Integration tests are designed to discover potential defects in the interaction of the individual components and their interfaces. In order to perform an integration test, two components are connected (i.e., integrated) and activated by appropriate integration test cases. Fig. 2 shows a schematic example of an integration test involving two classes.

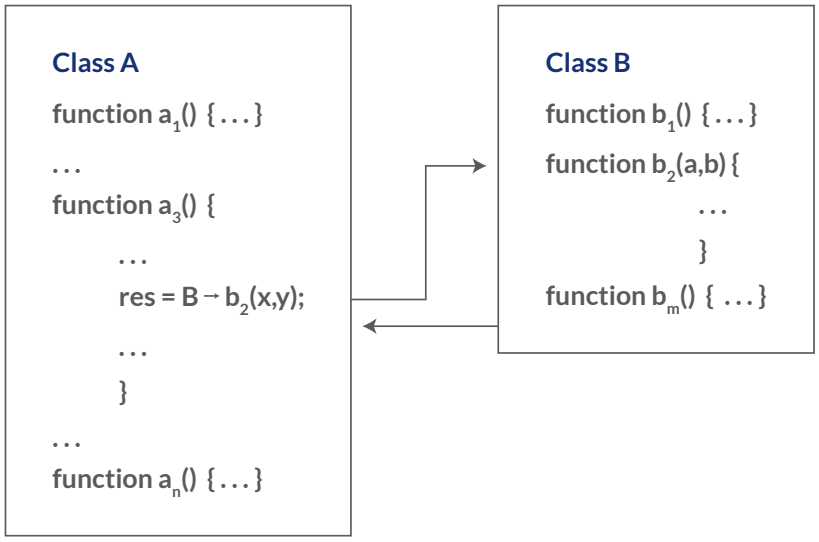

Figure 2. An integration test involving two classes

The interface of Class A is comprised of methods a_1, a_2, ..., an, while Class B comprises the methods b_1, b_2, ..., bm. Both classes have passed all their corresponding unit tests. The integration tests that we will now define test whether A and B work together properly. In our example, the interaction takes place between method b_2 and its call from method a_3[1]. We thus need to check whether b_2 is called correctly and whether it then returns the desired result to A.

TYPICAL INTEGRATION FAILURES AND THEIR CAUSES

Although A and B function correctly on their own, there are various defective behaviors that can turn up when they work in concert. The most important of these are interface failures:

1. Class A requires an object from class B to make it executable. In other words, A is dependent on B.

- A calls the wrong method from B:
  If we assume that A is a class within the eHome Controller's user interface and B is the class that represents individual devices, to switch off the kitchen lamp A would have to call the method switch('kitchen', 'lamp', 'off') in B. However, A has been wrongly coded dim('kitchen', 'lamp', '50').

- A calls the correct method from B, using invalid parameter values:
  As an example, A is meant to switch on the kitchen lamp but calls switch('living_room', 'lamp') instead. The first parameter is semantically incorrect, while the third parameter is missing entirely.

- The paired components code the returned data differently:
  In A, the call dim('kitchen', 'lamp', '50') means that the lamp is dimmed to 50% brightness using a value coded with values between 0 and 100. However, the dim() method in B expects the brightness value to be coded as a floating-point number between 0 and 1 (in this case, 0.5).

- A calls the desired method correctly but at the wrong time or in the wrong sequence (see section 4.1.3).
  As an example, the required object in B doesn't yet exist at the moment A accesses B and causes a runtime error in A. Even if the object in B already exists, it could be that one of its variables has either not been initialized or has been initialized with an incorrect value. This results either in a runtime error in B or, because of the erroneous variable, in B returning an incorrect value to A. Once A begins to work with an incorrect value, further failures are inevitable.

If the components communicate asynchronously[2], The following additional failures can also occur:

- Timing failures
  - A hands over data when B is not ready to receive it.
  - B returns a result to A either too early or too late. A cannot receive or process the returned data.
  - A fails to recognize the timeout or reacts wrongly to it (for example, by repeating the call).

- Throughput or capacity failures
  - A hands over more data than B can handle within a specific period of time, resulting in lost data or data overflow in B.
  - B returns more data than A can process.

2. Unlike in a synchronous communications loop in which the components involved wait for a response from the other side before sending or receiving data, the components in an asynchronous loop send and receive data without waiting for a response, reading and writing data to and from a shared data buffer.

- Performance problems
  - A hands over data faster or more often than B can handle and ends up waiting or re-sending data. The result is that processing speed between the two components decreases to below the specified minimum.

If the two components are distributed across separate hardware systems, transmission failures can also occur:

- The connection (the local network, for example) is damaged or down, resulting in the transmission of false, garbled, or no data.

Causes of failures:
If the components concerned were developed by different programmers or teams, the risk of integration defects increases. In such cases, the root cause of the defect is usually misunderstanding or differing interpretation of the specification or a simple lack of communication between the responsible developers. Changes to code and software updates can also cause defects—for example, if one component is altered but another that depends on it isn't. The compilers in compiler-based and statically typed programming languages usually detect these types of syntactical interface defects, whereas many interpreter-based languages (PHP, for example) only detect such defects at runtime.

eHome Controller Case Study 1:
Integration defects due to incomplete refactoring

The method set_status() in the Device class has been extended to include the write() call that writes an appropriate string to the data-base buffer when a valid change in status takes place. However, the person programming the database interface has renamed the class writeMsg() but forgotten to rename the method call in the Device class accordingly. Because the programmer has switched off the PHP interpreter's error messages, this defect initially remains undiscovered and only comes to light during the nightly integration test run when the write() call causes a runtime error.

Case Study 1.

Due to the continual alteration and improvement of the code that agile processes involve, the risk of code changes or updates causing integration defects is particularly prevalent in an agile environment. Therefore, even if just a single component is altered, you should always re-run not only its unit test but also all the integration tests that cover components that it interacts with.

DESIGNING INTEGRATION TEST CASES

Integration test cases are designed to detect defects of the types listed above. If components A and B are integrated in such a way that A uses B, A requires test cases that trigger all relevant usages of B. The corresponding test setup then looks like this:

3.2 TRADITIONAL INTEGRATION STRATEGIES

Traditionally managed projects usually perform integration and integration testing after unit (component) testing (see fig. 2-3). The assumption is that all software components have already been implemented and successfully unit tested before the integration phase begins. The integration team then takes subsets of grouped components, installs them in the integration environment and tests them. If the tests are successful, the subsystem in question is classified as integrated. If the tests fail, the defective component has to be reworked.

Although the system architecture determines which components belong to which subsystem, it is theoretically possible to organize the integration sequence as you wish. According to [Spillner/Linz 14], this can take place using one of the following basic strategies:

- Top-down Integration:
  Integration begins with a component that calls other components but is itself called only by the operating system.
- Bottom-up Integration:
  Testing begins with elemental components that do not call any others (except for operating system functions).
- Ad-hoc Integration:
  Components are integrated as and when they are completed by the programmers.

Pure top-down and bottom-up integration techniques require the system architecture to be designed on the basis of a tree diagram. The more crosslinked the basic architecture, the more you will have to deviate from the basic integration strategy. Additionally—as also found in projects managed using the V-model—it is rare for all components to be ready and unit tested, and integration thus often begins while some components are still being implemented. This inevitably leads to the use of an ad-hoc integration strategy.

3.3 CONTINUOUS INTEGRATION

Scrum-based projects continually produce new or altered code components, so we have to consider how to deal with them. A strategy based on traditional methods demands that the team waits until all code-related tasks are done and then installs all changes made by all programmers in the test environment for integration testing at the end of the Sprint. This is, however, a clear contradiction of the basic Scrum objectives:

- Integration test feedback to the programmer is unnecessarily delayed. In the worst possible case, a programmer who changes some code on the first day of the Sprint has to wait until the last day of the Sprint to receive integration test feedback.

- Because integration tests can and do reveal defects, the team has to include the time taken to remedy them in its plan. This forces the Sprint to adhere to a fixed sequence of coding, unit testing, integration (testing) and defect correction.

The result is a "Water-Scrum-Fall"—a strictly phase-oriented cascading approach to the Sprint. To prevent this from happening, the team needs to use a better integration strategy, namely: Continuous Integration. Continuous Integration (CI) is the next logical step in the development of an incremental integration strategy. Incremental integration means that every piece of code is installed in the integration environment and is integrated as soon as it is finished. New components are therefore not grouped with others that have not yet been integrated but are instead integrated in place of a previously integrated version within a central integration environment.

THE CI PROCESS

Alongside its central integration environment, the other most important aspect of CI is that it is fully automated. According to [Duvall et al. 07], the Continuous Integration process consists of the following steps and elements:

- Central code repository:
  The team manages its program code and automated tests in a shared central code repository. Code is version-managed and every programmer has to check his/her code into the repository either daily or, better still, after every change (i.e., as often as possible).

- Automated integration run:
  Checking code into the repository automatically triggers an integration process that takes place on a dedicated CI server and consists of the following steps:

- Compilation
  The code is compiled, and compiler warnings and error messages are logged on the CI server. Compilation runs usually take a number of seconds.

- Static code analysis
  Successful compilation is followed by static code analysis that checks that the team has stuck to its own coding guidelines and quality metrics. These results are also logged on the CI server. Here too, a typical run takes a number of seconds. It is essential that the team only uses coding guidelines that can be checked automatically. No guidelines are to be drafted for code attributes that cannot be checked automatically or ones that the team considers to be insignificant.

- Deployment to the test environment
  Once the code has been compiled and statically checked, it can be deployed to and installed in the appropriate test environment, which is reset to a predefined state first.

- Initialization
  All initialization steps—such as creation and filling of database tables—take place automatically. The test data required for the tests that follow is also imported at this stage.

- Unit testing
  Automated unit tests for all units follow. The results are logged on the CI server.

- Integration testing
  Next up are automated integration tests. The results are logged on the CI server.

- System testing
  Finally, automated system tests are started. These results, too, are logged on the CI server. System tests can last up to several hours and are therefore not part of every CI run. They are usually run daily, normally during the night.

- Feedback and dashboard:
  The CI server displays all results on a central browser-based dashboard that gives immediate feedback in the case of test failures or other issues. Feedback is also pushed actively to the programmer(s) concerned via e-mail or other means.

Figure 3 (after [Duvall et al. 07], with additional test environments) shows a schematic of a typical CI environment.

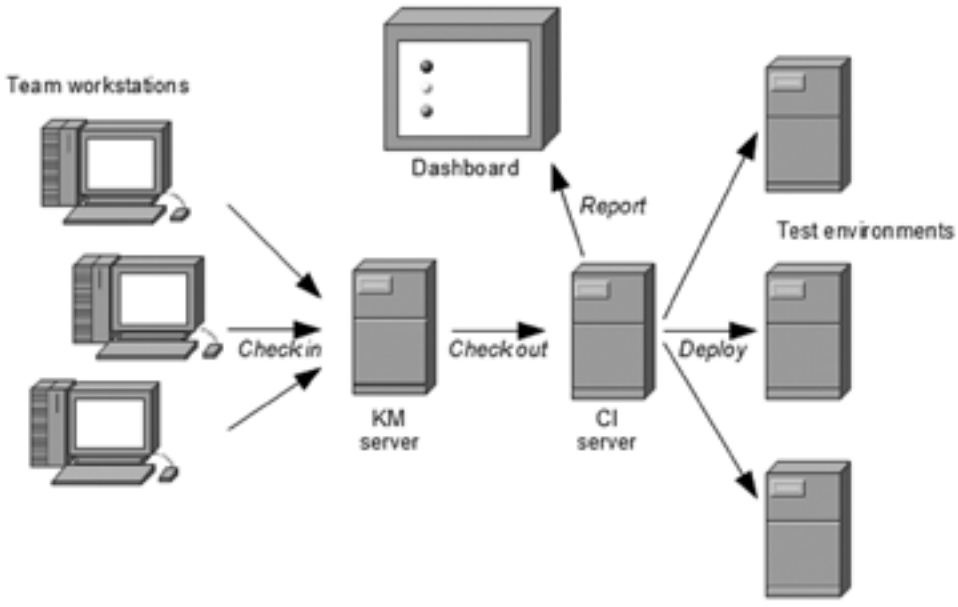

Figure 3. A CI environment

The CI dashboard:
The results of each step are displayed on the dashboard as they occur, not at the end of the complete CI run. The speed of the feedback loop to the team depends on how long each CI step takes. Compilation and static analysis typically take seconds

or minutes to complete, while unit and integration testing usually takes a number of minutes. In contrast, system testing usually takes several hours. The CI server logs all test results and displays them in summarized form on a dashboard, keeping the entire team informed of the state of each CI run and the overall quality of the system. Figure 4 shows the dashboard used by the testBench team (see case study 8.2).

Test Statistics Grid							
Job	Sucess #	%	Failed #	%	Skipped #	%	Total #
iTB Integration Tests	970	100%	0	0%	0	0%	970
iTB Packaging	0	0%	0	0%	0	0%	0
iTB REST Tests	18	100%	0	0%	0	0%	18
iTB Static Analysis	0	0%	0	0%	0	0%	0
iTB Nightly 2	662	<100%	2	>0%	0	<0%	664
iTB Utilities	1320	100%	0	0%	0	0%	1320
iTEP Export Plugin	5	100%	0	0%	0	0%	5
iTEP4	548	100%	0	0%	0	0%	548
ITORX	5	100%	0	0%	0	0%	5
Word Reporting	163	98%	0	0%	3	2%	166
Total	3691	<100%	2	>0%	3	>0%	3696

Test Suites Overview

Failed (0%)	Passed (100%)	Skipped (0%)					
iTB- unit-tests			0	997	0	997	100%
iTB-integration-tests_ORACLE			0	2936	0	2936	100%
iTB-integration-tests_MSSQL			0	2936	0	2936	100%
iTEP tests			0	548	0	548	100%

Figure 4. A CI dashboard, showing some typical results

In agile projects, system testing is usually much less automated than unit and integration testing[3]. System testing usually involves a fair amount of manual intervention that can take a number of days to complete. Because each Sprint generates new features that require their own system tests, the number of manual system tests grows from Sprint to Sprint. Even after just a few Sprints it becomes clear that the team cannot perform all its system tests for every Sprint the way it does for its unit and integration tests, and even less so for every change in the code. The team then has to carefully plan when to perform which system tests. The following sections describe a number of approaches to system test timing.

4.1 SYSTEM TESTING IN A FINAL SPRINT

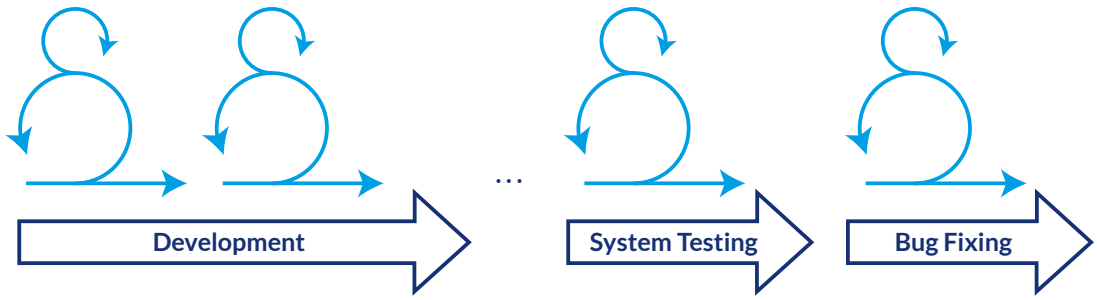

Figure 5. System testing in a final Sprint

This is the simplest system test timing strategy, and involves planning a separate "system test Sprint." This approach is easy to plan and can be used for exclusively manual system tests. The system test Sprint contains only system testing tasks. The drawbacks of this approach are:

- You can only tell if you have a potentially shippable product once the system test Sprint has been completed. If the system test Sprint is still pending, shipping a product involves a high level of risk.
- The amount of bug fixing and retesting effort involved can only be estimated once the system test Sprint is complete, and can involve additional bug-fix or retest Sprints.
- System testing feedback to the developer only takes place after the final system test Sprint, which is often much too late. The aim of providing timely feedback to the programmers is not met.

Independent system testing team:
The overall procedure is very similar to traditional V-model system testing and falls short of utilizing the advantages of agile practices—especially the creation of a potentially shippable product after every Sprint.

3. In traditionally managed projects, these proportions are usually reversed. The independently organized system testing group builds a comprehensive, automated GUI test system, while programmers who have little QA and testing experience are only marginally involved in creating unit tests.

One way to work around these disadvantages is to form an independent system testing team. (Case study 8.5, "Scrum in a Medical Technology Environment," describes a good example of this approach). The new team can shoulder the system test Sprint while the product team has more capacity to push ahead with the next development-related Sprint. This way, development and system testing can run in parallel at a one-Sprint offset. Bugs that are discovered in the course of a System test Sprint can be debugged during the next development Sprint and, from a programmer's point of view, the feedback loop is only two Sprints long.

4.2 SYSTEM TESTING AT THE END OF A SPRINT

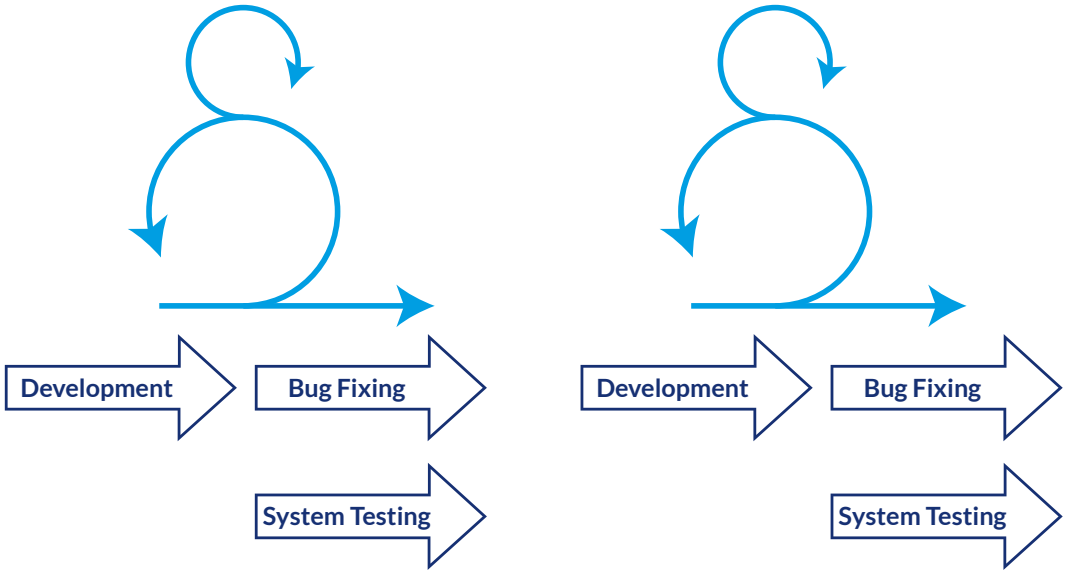

Figure 6. System testing at the end of a Sprint

This strategy involves performing system testing in a specially reserved slot at the end of each Sprint, and is suitable for use in simpler projects that do not require a lot of system testing. If the amount of system testing effort threatens to expand too far, the team has to select which system tests to perform and which to leave out of each Sprint. In this case, it is important for the test manager to keep an eye on the situation, which an increasing degree of system test automation will help to improve. The disadvantages of this strategy are:

- As with the system test Sprint strategy, this approach also produces an unknown quantity of bug fixing and retesting tasks at the end of the Sprint, thus endangering the planned timebox.

- To gain time for bug fixing tasks while sticking to the planned timebox, User Stories have to be regularly removed from the Sprint Backlog.

- Because time is limited and only selected system tests can be performed, the risk of overlooking defects and carrying them over into subsequent Sprints increases. This risk is less severe if you perform regression testing for all system test cases.

- A programmer has to wait until the end of the Sprint to receive feedback. If the corresponding system test cases are included in the done criteria for a programming task (as it should be), these tasks remain "undone" until the end of the Sprint, making it impossible to effectively estimate project velocity.

Sprints become serialized like a waterfall.
Feedback to the team takes place within the Sprint although the tasks are serialized like a waterfall. Because system testing always reveals some defects, a certain amount of time has to be reserved for bug fixes and retesting. The Sprint is therefore inevitably divided into programming, system testing, and bug-fixing phases—a situation often referred to as a "Water-Scrum-Fall."

4.3 SYSTEM TESTING NONSTOP

The system testing nonstop strategy applies the principles of unit and integration testing to system testing, and is probably the best solution. "System testing nonstop" means automating system tests as far as possible and integrating them in the automated CI process like unit and integration tests. According to the required preconditions and run times, system test cases can be grouped into suites that can be run or hidden, and the resulting CI environment includes all the project's tests, from unit to system level.

An interview with Joachim Hofer, TestBench Development Manager and Dierk Engelhardt, TestBench Product Manager at imbus AG

imbus AG is a German company specializing in software quality assurance and testing. At the time of writing, imbus had more than 200 employees working at locations in Germany, Sousse/Tunisia, and Shanghai, China. The company offers consulting, software testing and test outsourcing services as well as its own testing tools and training. Its customers are software manufacturers and the software development departments of government agencies and companies in all branches of industry.

TestBench is a powerful test management tool developed by imbus. It covers all aspects of the testing process, from planning, design and automation to test execution and reporting of all software testing tasks. TestBench is used by customers in medical technology, railway engineering, the automobile industry and in banking and insurance.

The tool was developed using Java by a team comprising between 12 and 16 members. The product is implemented and integrated at customer sites by specialist TestBench consultants. Until 2010, the product was developed using an iterative, phase-oriented process with programmers and testers working in separate groups and producing one or two major releases per year.

IMPROVEMENT GOALS

This development process caused the typical issues that phase-oriented models are known for. System testing only began once implementation was complete and programmers were already celebrating the completed iteration. The system testers regularly dampened the programmers' enthusiasm with a series of defect reports. From the programmers' point of view, these reports came unnecessarily late and often addressed issues that required intervention for which there was no additional time available in the current iteration. Each iteration was therefore separated into an implementation phase and a subsequent bug-fixing phase, with the latter taking place under enormous pressure because of the limited time available and the need to succeed. The result was a high-quality, stable product but the development process was always an uphill struggle.

To alleviate this situation, the goal of parallelizing programming and testing was set. The idea was to provide more timely testing feedback to the programmers and further automate the testing process to provide greater code refactoring safety for the future of the product's development.

INTRODUCING AGILE DEVELOPMENT TECHNIQUES

Starting in 2010, development manager Joachim Hofer began introducing a whole raft of agile practices in order to reach these goals:

- Requirements management
  Previously, a requirement took the form of a headline in the Caliber requirement management tool that was linked to a detailed Word document. Implementation could only begin once the entire document had been approved. The decision was made to move away from large requirements documents and instead to rely on smaller User Stories built up successively and logged using the Jira issue management tool.

- Nightly build
  The rules for the production of nightly builds were also tightened. A central compilation and integration environment already existed and the programmers logged in their code whenever it was ready (on average, every two or three days). This approach created a certain amount of unfinished code that was not yet checked in, so the decision was made to have all programmers check in all their code every evening. The existing automated unit tests were then performed on the complete code, which initially led to all sorts of problems. However, the result was an important learning experience that taught the team to work daily toward finished, executable code. The packets of code that the programmers tackled each morning became smaller and the planned changes were mostly complete by evening.

- Nightly automated system tests
  In addition to further automating unit testing, the team pushed ahead with the automation of system testing too. The test environment was extended so that the unit and integration tests were followed by nightly automated system tests started directly from the build environment. All new system tests were designed immediately and implemented using the QF-Test tool so that they could be included in the nightly test environment.

- Continuous Integration
  Originally, each build took four hours to produce, which was just doable in the context of a nightly build but still too slow to ensure that the nightly tests were complete before work began the next day. The build time was reduced to 15 minutes by separating each build into a series of sub-projects and rebuilding the build environment as a Jenkins/ Hudson environment. All automated unit, integration and (nightly build) system tests were repackaged and embedded in the improved build environment. Depending on the test package being run, feedback times were reduced to a minimum of 15 minutes—i.e., no longer than a coffee break!

- Static code analysis and coverage measurement
  The dynamic unit tests in the continuous integration (CI) environment were extended to include additional static code analysis that runs parallel to automated integration testing. For example, the FindBugs tool is used to identify typical Java coding issues such as incorrect API calls.

- Task orientation
  To achieve sufficiently granular task control within each iteration, taskoriented working methods based on User Stories were introduced. To prioritize tasks, a scoring system was introduced that evaluates task priority using the customer's viewpoint, the number of corresponding customer requirements and an internal vote within the team.

- Daily Standup
  The team now meets for 15 minutes every morning. Each team member reports on what he/she is doing right now, how work is progressing and on any current issues. Everyone decides for themselves which tasks to report. As yet, there is no real Sprint Planning to compare with the reported tasks, but the daily nature of the meeting makes great practice for a forthcoming Daily Scrum.

All these practices were introduced while preserving the original iterative development model. However, the use of User Stories and continuous integration (CI) went a long way toward dissolving the strict phases of the earlier development process and helped to interlace and parallelize them. The next step involved introducing agile product and project management practices and transforming the team into a self-organized unit.

INTRODUCING SCRUM

The new development techniques listed above were introduced largely by development manager Joachim Hofer. The time had now come to involve the whole team in the Scrum process. Joachim Hofer and product manager Dierk Engelhardt decided to make the switch in the course of several individual steps:

- Research and inform
  Kanban, Scrum, XP and other agile methodologies were considered, but the discussion and the planning that followed quickly focused on Scrum. An internal forum was set up in which the team could add its thoughts and make suggestions regarding how to set up the new Scrum-based development process. Spurred on by their own curiosity, all team members studied current Scrum literature and visited websites such as scrum.org to get an idea of what Scrum is all about. This process was accompanied by internal workshops and regular team discussions.

- Restructuring the team
  The switch to Scrum of course required a change of roles, responsibilities and tasks within the team. Joachim Hofer became the Scrum Master, the test manager became his deputy, and the product manager took on the role of Product Owner. The division between testers and programmers was abolished and pair programming was introduced with tester/developer pairs usually responsible for testing and integration tasks, pairs of two developers for development tasks and pairs of testers for system testing tasks.

- Sprints
  There is no ideal or easy moment to give up your old development habits and start the first Sprint—you simply have to go for it. At imbus, the first Sprint Planning meeting took place on a Monday early in 2011. The Product Owner had already transferred what he considered to be the most important requirements from the old project plan to the Project Backlog, which the team turned into a set of task cards for a fourweek Sprint during an initial one-day Sprint Planning session. The team later switched to a three-week Sprint cycle.

MAJOR CHANGES

The dissolution of the differences between the traditional roles of testers and programmers and the adoption of the role of all-round team members was generally acknowledged as the most serious change. The introduction of pair programming played an important role in making the transition successful.

- Pair programming made code reviews a regular part of the workflow. Every programmer hands over new code to his/her partner for review as a matter of course. This practice also stimulates the sharing of knowhow within the team.

- The milestone/work package approach to task planning was replaced by a task-oriented approach to task management. This reduced the importance of the Caliber requirement management tool and increased the importance of the Jira issue management tool and the Greenhopper plug-in (for Backlog management, task ranking, task board management and metrics/charts).

- The main Scrum tools Backlog, Sprint Planning/Planning Poker, timeboxing and Retrospectives were successfully introduced and have been practiced since in a sustained and disciplined fashion.

- User Stories enabled the introduction of stricter development rules. From the very start, the team has had to write test cases for every code change or new User Story in Jira. Previously, team members were allowed to write tests once a change in code had been completed. Now, a programmer/tester pair is responsible for writing unit, integration or system tests for each User Story in advance. The programmers then use the TestNG tool to implement and execute unit and integration tests, while the testers automate the system tests using QF-Test.

- The introduction of CI enables parallelization of programming and testing tasks as planned. Every time a change in code is checked into the system, a build run results. This consists of a minimum of compilation and unit tests and it takes about three minutes for feedback to reach the programmer. Integration tests lasting 15-30 minutes follow.

- In addition to the CI runs that are triggered by the programmers' code check-ins and take place several times a day, the team also makes a nightly build. Each run involves starting a virtual machine with a freshly installed operating system and installing the last successful version of the product to emerge from the CI process. Automated system testing controlled by TestBench follows, and currently comprises about 15,000 data-driven test steps that take approximately 10 hours to complete. The code is automatically instrumented in advance and currently achieves about 40% line coverage. Video captures of the nightly system tests are made that enable staff to follow failed tests visually the next morning. Total code coverage within the CI environment is about 60% and reaches almost 100% for newly implemented features or User Stories. Older code for which no automated tests exist unfortunately reduces overall coverage.

- The test specifications used previously were largely replaced by commented test code and, for the system tests, by keyword-driven, machine-executable test specifications. TestBench is closely linked to Jira and the Jenkins environment and is used to successfully manage all system tests.

- The system tests are not completely automated and manual testing will, in future, still be necessary for checking report graphs, usability, etc., so an additional half-day, session-based exploratory system test takes place at the end of each Sprint. The switch from iterative to agile development has nevertheless drastically reduced the manual testing effort from several person-weeks to one person-day every three weeks.

- Sprints are three weeks long and result in a tested internal product release. As previously, external releases take place twice a year. Changes in customer requirements can be handled right up to the start of the last Sprint (i.e., as little as three weeks before delivery).

LESSONS LEARNED

- The introduction of agile development techniques (such as continuous integration) before we switched to using Scrum gave us a stable base on which to build all of our Sprints right from the very start.

- Setting up the tool infrastructure (mainly for CI) involved significant effort that shouldn't be underestimated. Setup work definitely reduces the potential feature productivity of the first few Sprints.

- Pair programming is crucial to the team's success, although some team members find it easier to work with certain colleagues and do not harmonize with everyone. Pairs have to be allowed to form themselves and cannot be forced to work together.

- Test-driven development improves the overall code architecture, and the number of defects that have to be managed has been significantly reduced.

- Regular Sprint Retrospectives deliver a continuous stream of ideas on how to improve things (e.g., improved effort estimation or discussion of questions like “What exactly is a Story Point?”).

- Scrum does not reduce the overall workload and does not create additional resources! However, work that is done really is finished. We no longer have to deal with large numbers of defects and painstaking rounds of bug fixing.

- Product roadmaps are replaced by strategic planning.

- Switching to Scrum requires commitment on the part of the team managers and the team, and takes time. It also requires research, training and the setup of new infrastructure that cannot be accomplished overnight.

CONCLUSIONS

After a year’s experience with Scrum, Dierk Engelhardt and Joachim Hofer confirmed that they were able to reach the goals they set out to achieve. The team now uses important agile techniques, such as test-driven development and continuous integration, in a disciplined and sustained fashion. Their test-driven development and zero defect strategies are particularly effective and the integration of TestBench and Jenkins have made nonstop testing (i.e., continuous automated unit, integration and system testing) a reality. Comprehensive refactoring has also become much less of a risk and the effort involved in producing an external release has been significantly reduced. The team is really happy with the new arrangements.

However, the team isn’t content to rest on its laurels and the next improvements to the development process are already being planned. These include the use of Atlassian Confluence to describe the system requirements and further acceleration of the CI process through test parallelization and upgrades to the build and test server hardware.

rockynook

To purchase the full version of

'Testing in Scrum: A Guide for Software Quality Assurance in the Agile World'

Click Here

www.eurostarconferences.com

Join us online at the links below.

EuroSTAR
Software Testing Conference