TESTING STRATEGIES AND METHODS

Game Changing Test Methods

Michael Hackett

Risk Based Testing, Strategies for Prioritizing Tests against Deadlines
Hans Schaefer

Something Old, Something New: Requirements and Specifications
Allan Kelly

5 Tips for Getting Software Testing Done in the Scrum Sprint
Clemens Reijnen
I spend about half my work time in the role of a consultant assessing, auditing and examining software development team practices and processes for the purpose of process improvement.

I am regularly surprised to find teams that lack basic skills, management support, tools, information, access to users, Product Owners and to developers. And yet they’re still expected to carry the heaviest burden of quality and customer satisfaction for the company. It’s the lack of easy access to skills and lack of input on team’s process that is the most disheartening.

Most testers/test engineers/quality engineers have a real desire to be the best they can be at their job; find great bugs, be a team player through project problems and create satisfied customers. Unfortunately, many people who get into testing come from various backgrounds with little to no prior formal testing training. The most common training is on the job “doing what we have always done.” This is where I came in.

There are real project and quality issues facing test teams these days. Teams are dealing with all kinds of problematic issues like Development “handoffs” to test teams while still calling themselves Scrum teams, lack of quality practices and methods, lacking automation, or lacking effective automation trying to have Continuous Integration (CI). It’s common to find teams who have never been trained in scrum using totally scrumbut practices—double standards where the documentation load has been reduced on other team members while testers must still document every test and write large test plans. If you think “You may be a Scrumbut,” check out this great description from the Scrum Alliance: https://www.scrumalliance.org/community/articles/2013/february/you-may-be-a-scrum-but

Many test teams need help with the basics. There are also rare occasions where efficient, effective teams need help moving to the next level, for example, teams that have graduated from Scrum and moved to Kanban for their SDLC. My best recommendation in most situations I have described above is to learn as much as possible and strive for incremental change. Change one thing at a time. Change how you do things, learn one new thing and share it with the team.

Our job at LogiGear magazine is to provide you with quick access to high quality great new ideas for testing and quality. In this issue we present you with two rising test methods, BDD and ABT. We also have great pieces on strategically managing risk and effective requirements analysis. And Clemens Reijnen writes a very timely article on testing in Scrum. We have also done methods and strategy issues in the past, the last from just a year ago, February 2014. Our fully searchable archives are full of great articles on various test methods that you can apply from mobile testing to test automation.

From all over the world- we seek out stories and thought work to best benefit teams thrive today. We want to provide one-stop shopping for testing and quality improvement. It is my earnest hope that you can learn something to help you and your team in this issue and have a positive effect on your job satisfaction, daily work, team, product, and customer satisfaction. Good luck!
IN THE NEWS

COVER STORY

5 BDD AND ABT: Game Changing Test Methods
Two powerful test methods that can make Agile teams faster
Michael Hackett

HOW TO: ABT TEST MODULE TEMPLATE

10 Template for Creating ABT Test Modules
A top-down planning approach helps to create test cases that are free of unnecessary details and redundant checks.

ABT: MORE THAN KEYWORDS

11 ABT and Keyword-driven Testing - Similarities and Differences

ARTICLES

Risk Based Testing, Strategies for Prioritizing Tests against Deadlines
When it is out of the to delay delivery the solution is a prioritization strategy in order to do the best possible job within the time constraints.
Hans Schaefer

Something Old, Something New: Requirements and Specifications
Differences in interpretation of requirements and specifications by programmers and testers is a common source of bugs.
Allan Kelly

5 Tips for Getting Software Testing Done in the Scrum Sprint
A clear practice with the support of tools will help teams be more successful in delivering done products.
Clemens Reijnen

BOOK REVIEW

Integration Testing from the Trenches
The concepts of integration testing presented with a casual formalism that's intuitive and fun.
Victor Grazi
A CASE FOR BETTER SECURITY TESTING

According to a report by Autoblog, a 14-year old boy, who was part of the 2014 CyberAuto Challenge, was able to remotely hack into a car with nothing but a handful of parts from RadioShack and an iPhone. What’s worse is that he was able to crack the vehicle’s security systems in one day, with materials that cost just $15.

Like something out of the Watch Dogs video game, the boy in question (who has not been named) collected the parts and spent the night assembling a homemade circuit board. The next day, he was able to operate the vehicle’s wipers, lights, door locks, and even the remote start feature. He even played a song from his phone through the car’s speakers, flashing the headlights to the beat in a clever taunt.

Source: http://www.digitaltrends.com/cars/14-year-old-hacker-breaks-into-car/

HANS BUWALDA TO PRESENT AT STAREAST MAY 4

LogiGear CTO Hans Buwalda will share his experiences and present strategies for organizing and managing testing on large projects at STAREAST. He will present how to design tests specifically for automation, including how to incorporate keyword testing and other techniques. Also covered is what roles virtualization and the cloud can play—and the potential pitfalls of such options. He will also give tips and tricks to make automation more stable, and to deal with the numerous versions and configurations common in large projects.

iOS PUBLIC BETA AVAILABLE FROM MARCH

After its successful public beta program for OS X Yosemite, Apple is said to expand the program that would mean forthcoming versions of its mobile operating system, beginning with iOS 9, would be subject to the same assessment. The iOS public beta release highlights just how open Apple is becoming. Apple has already allowed a million members of the public to sign up to test OS X Yosemite, and that might be the same approach that it takes with iOS 9. Of course, anyone that signs up to Apple’s developer programs can road test the betas, but that’s not a free program and requires a yearly membership.

Source: http://www.streetwisejournal.com/apple-ios-public-betas/7033/
As development teams have been pushed faster and into tighter scrum sprints, testing has burst through old development paradigms. Developers are being pressed to do more unit testing. Automated smoke tests are essential parts of CI (continuous integration) and full, automated regression suites are being run across platforms, devices, virtual machines. With all the complexity, many teams have finally come to the conclusion that having a more methodological approach and strategy for test design, execution and maintenance is essential to keep up with the speed demands.

Testing is a dynamic business. Testing methods are developed and turned into acronyms that are tossed around as if everyone knows and understands them. After the initial hype most fizzle out. However, two methods in particular have proven valuable, and are popular and growing in use—Behavior Driven Development (BDD) and Action Based Testing (ABT).

The BDD and ABT methods

BDD is great. ABT is great. Both methods have been used extensively and undergone refinement over time. It's important to understand that they are not interchangeable as test methods. One is not *better than* the other; the methods are quite different with different users, and different goals. To make effective use of either method requires fully understanding what each method is best suited to do and use it for that.

BDD was pioneered by Dan North in 2006 and grew out of Test Driven Development (TDD). ABT was pioneered in 2001 by Hans Buwalda and it grew out of keyword-driven testing. Each method is very good for what it is meant to do. Some situations are better suited for one than the other as they are practiced by different users for different purposes. This highlights a big problem in having software development as a sophisticated area of technical expertise: many people lack an understanding of various test methods with which to strategize a fully effective test project.

What is BDD?

Behavior driven development was developed out of a need for a guideline for effective and efficient TDD (test driven development), the eXtreme Programming practice. What is TDD? See TDD SideBar on page 6.

TDD is an idea, or paradigm. BDD is one way to execute it, and it's a good way. It gives easy direction to programmers on how to do product specification and unit test with a low learning curve. Its purpose is to help the folks devising the system (i.e., the programmers) identify appropriate tests to write -- that is, tests that reflect the behavior desired by the stakeholders. It involves the business and test teams through writing tests based on behaviors.

The main focus of BDD is to provide a clear understanding of software behavior by making use of natural language while writing test scripts.

My response [to TDD] is behavior-driven development (BDD). It has evolved out of established agile practices and is designed to make them more accessible and effective for teams new to agile software delivery. Over time, BDD has grown to encompass the wider picture of agile analysis and automated acceptance testing.

from Dan North “What’s in a Story?”
What Is ABT?

ABT is a method based on keyword-driven testing that is focused on intelligent test design and organization that uses the concept of test modules to increase the efficiency of test development. Test modules provide a level of abstraction over test cases and make it possible to create well-defined test case flows. The top-down planning approach helps to create test cases that are free of unnecessary details and redundant checks. ABT also effectively separates the design of the tests from how they will be executed, allowing tests to be used for both manual and automated test execution.

While ABT has similarities, it is not the same as Keyword driven testing (see page 11)

Action Based Testing is an advanced keyword-driven test method that places emphasis on test design ahead of automation and execution. ABT encapsulates modular test design and action-driven methodology that makes large-scale, collaborative test development possible.

Hans Buwalda

In ABT, test cases are authored using readable keywords, or actions to symbolize functionality to be tested. It allows subject matter experts to create well defined tests and a single automation engineer to support a large group of testers by automating the execution.

Contrasting BDD and ABT

The two methods have a number of good aspects in common. Both ABT and BDD focus on business case driven behaviors for testing. They facilitate writing tests using good design practices without being concerned about how the tests will be executed—execution follows.

They are both written in a natural, domain specific, language that is not code and is easy for all people on the team to read and comprehend. Both methods:

- have some structure, but it’s not rigid
- have loose syntax
- are meant to be short and simple statements
- separate out test data
- focus on the separating test design from test execution
- can be used before any code is written, more Agile
- BDD and ABT allow automation to be done later.

About TDD

Test-driven development (TDD) is a software development process that relies on the repetition of a very short development cycle: first the developer writes an (initially failing) automated test case that defines a desired improvement or new function, then produces the minimum amount of code to pass that test, and finally refactors the new code to acceptable standards. Kent Beck, who is credited with having developed or ‘rediscovered’ the technique, stated in 2003 that TDD encourages simple designs and inspires confidence.

Test-driven development is related to the test-first programming concepts of extreme programming, begun in 1999, but more recently has created more general interest in its own right.

Programmers also apply the concept to improving and debugging legacy code developed with older techniques.

http://en.wikipedia.org/wiki/Test-driven_development

There are two levels of TDD:

1. Acceptance TDD (ATDD). With ATDD you write a single acceptance test, or behavioral specification depending on your preferred terminology, and then just enough production functionality/code to fulfill that test. The goal of ATDD is to specify detailed, executable requirements for your solution on a just in time (JIT) basis. ATDD is also called Behavior Driven Development (BDD).

2. Developer TDD. With developer TDD you write a single developer test, sometimes inaccurately referred to as a unit test, and then just enough production code to fulfill that test. The goal of developer TDD is to specify a detailed, executable design for your solution on a JIT basis. Developer TDD is often simply called TDD.

http://agiledata.org/essays/tdd.html#WhatIsTDD
One very important thing to understand about these methods is that they prescribe a top-down approach to test development. For most people this means there will be a steep learning curve, and for most projects more effort will need to be expended on up-front planning. For those who are used to equating code production with progress, this will seem counterproductive—at first. The end result will be that test case creation will be faster as they will be better thought-out and simpler. In ABT’s case, this also minimizes the effort to maintain tests.

Can you focus BDD and ABT only on Test Automation? Sure. It’s common. Some people think these are only methods for automation, but focusing only on test automation misses the full power of these methods. BDD and ABT are powerful methods to make sure you are building the right thing and have the ability to design tests early, at the beginning of a sprint, and leave the action automation to an automation engineer later after that functionality has been built.

Let’s look at constructing BDD and ABT tests to see the work products and what it looks like to see the full power of BDD and ABT and how different they are.

**Writing BDD tests**

One of the most initially appealing things about BDD is the simplicity of the documentation which is produced using a common natural language, most commonly Cucumber. [http://en.wikipedia.org/wiki/Cucumber_(software)]

A programmer reads a spec, requirement, or user story and then tries to describe the behavior before writing code. Dan North suggests starting with scenarios; how he defines scenarios is more commonly called a test objective.

BDD tests follow the format of a Scenario title for the test followed by a Given-When Then (GWT) syntax describing the product behavior. [http://dannorth.net/whats-in-a-story/]

Title (one line describing the story)

As a [role] I want [feature] So that [benefit]

Acceptance Criteria: (presented as Scenarios)

Scenario 1: Title
Given [context]
And [some more context]... When [event] Then [outcome]

And [another outcome]...

Scenario 2: ...
Below is an example of a BDD test scenario for cash withdrawal from an ATM

Story: Account Holder withdraws cash

As an Account Holder I want to withdraw cash from an ATM
So that I can get money when the bank is closed

Scenario 1: Account has sufficient funds
Given the account balance is \$100
And the card is valid
And the machine contains enough money
When the Account Holder requests \$20
Then the ATM should dispense \$20
And the account balance should be \$80
And the card should be returned

Scenario 2: Account has insufficient funds
Given the account balance is \$10
And the card is valid
And the machine contains enough money
When the Account Holder requests \$20
Then the ATM should not dispense any money
And the ATM should say there are insufficient funds
And the account balance should be \$20
And the card should be returned

Scenario 3: Card has been disabled
Given the card is disabled
When the Account Holder requests \$20
Then the ATM should retain the card
And the ATM should say the card has been retained

Scenario 4: The ATM has insufficient funds...

While this seems very easy, Gherkins can be, or can become, deceivingly complex.

When BDD is correctly used for TDD the test are straightforward and small since TDD focuses on small chunks of code. However, the natural tendency is to have larger tests. Twelve to fifteen lines of “Given—When—and—and—and—Then—but—but” are more common than not.

When trying to adapt BDD for UI testing (not the primary intention of BDD), the tests can get long and complicated, which increases maintenance. The addition of “and—and—and—, but—but” adds complexity to tests when simplicity is the goal.

Breaking tests down into small, manageable (and maintainable) chunks is a discipline that has to be practiced to learn.
**Writing ABT tests**

In ABT, anyone on the project can write test objectives that identify what has to be tested, and then detail the task or workflow using actions (keywords). This style test can be created quickly regardless of how the tests will be executed. Also, with ABT, adding tests built upon already defined tests and actions is easy, scaling test development rapidly to high volume suites.

ABT starts with defining test modules which prescribe what tests will be included in each module. This is followed by defining test objectives in a cause and effect syntax. The test objective describes what each test case is supposed to do. From the objectives it’s a straightforward process to detail the test case for each object with actions and the appropriate data to execute the test.

**Basic Syntax**

**Test Module**

Test Objective: Cause and Effect

Test Case 1

<table>
<thead>
<tr>
<th>Action</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Action</td>
<td>data</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Action</td>
<td>data</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Check</td>
<td>data</td>
</tr>
</tbody>
</table>

Test Case 2

<table>
<thead>
<tr>
<th>Action</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Action</td>
<td>data</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Action</td>
<td>data</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Check</td>
<td>data</td>
</tr>
</tbody>
</table>

...and so on.

The example at the right shows a functional ABT test of cash withdraw from an ATM that follows a basic ABT template. The test scenario has three test cases that require only five actions. The ABT template can be found here and on page 10.

When learning the syntax for writing ABT tests, the easiest part for everyone to grasp is the aspect of reusable actions to create low maintenance small tests. The part of ABT that is, for many people, the most effective is creating actions that can be reused and shared with teams. This is the keyword-driven aspect ABT was built on.
More examples of Test Objective syntax: *cause & effect format, short & descriptive.*
- Refunded items should be returned to stock
- Replaced items should be returned to stock
- "clicking submit empties all fields"
- "If all fields are populated, ok is enabled"
- "ok becomes enabled if both first name and last name are specified"
- "in the case of a sports car: the screen specifies seconds to reach 60MPH"
- "in the case of a sports car: no lease price is available"

**Best Uses**

ABT and BDD are different. They solve different problems and have different best uses. Here are some examples.

**Situation 1** – The scrum team does not fully understand the user stories or product. The user stories are not very detailed or the product has complicated domain specific tasks where the dev team may have only a high level knowledge of the workflow and domain. Use BDD to show behaviors of the system and get Product Owner buy-in. Big bonus for the team.

**Situation 2** – Programmers do no unit tests, testers do all automation and regression. Programmers can write Given-When-Then statements and use a common unit test harness, such as JBehave for automated unit tests. Big bonus for the team.

**Situation 3** – Test team lacks programming skill. Domain knowledgeable team (testers, POs, subject matter experts, business analysts) can write the various test objectives and conditions without concern for how the automation will get executed. An automation engineer using an action based automation tool, like TestArchitect, can automate and support about a dozen domain knowledge people. Big bonus for the team.

**Situation 4** – A highly skilled, fully Agile team. Programmers can use BDD to better specify the system behavior. Action based test modules and test objectives can be developed at the same time. Once coding starts and you have product to be tested, the BDD will be turned into TDD/unit tests. The ABT tests (bigger, longer, more complex tests) can be automated. Big bonus for the team.

**Situation 5** – Your product has multiple platforms or devices and you need a lot of high volume automation. Use ABT to design tests. Since ABT focuses on test design separate from the execution, the same tests can be executed across various platforms. An automation engineer, using a cross-platform tool, can change the implementation of the automation and the bulk of the work developing the tests might not change at all. Big bonus for the team.

**Situation 6** – High maintenance cost for test automation. If you have a product that is very often in a state of change and/or constant redesign, your tests are probably high maintenance. Building a reusable, high-volume automated regression suite using a test method focused on low maintenance and high reusability is a must. ABT which focuses on small action level maintenance is best for this. Big bonus for the team.

**Summary**

Test Methods are useful sets of practices for test design, communication, measuring coverage, execution, documentation, modeling, and various other aspects of the development process.

Better understanding BDD (primarily used by programmers) and ABT, (primarily used by test engineers) will help you choose the best test method for various situations. ABT & BDD are not interchangeable. While BDD is great to specify a product and build unit tests, ABT is great at build high volume, cross platform, robust automation suites. Each method is very good at what it is meant to do. Used correctly both methods will result in better testing.

Michael Hackett co-founded LogiGear in 1994 and leads the company’s training operations division, setting the standard in software testing education programs for many of the world’s leading software development organizations.

Michael also co-authored the popular Testing Applications on the Web (Wiley, 2nd ed. 2003), and has helped many clients test and deploy applications across multiple platforms and multiple language editions.
### ABT Test Module Template

Action Based Testing (ABT) is an efficient method of test development that provides a systematic approach to increase the success of automated testing.

ABT uses Test Modules to increase the efficiency of test development. Test modules provide a level of abstraction over test cases and make it possible to create a well-defined test case flow. The top-down planning approach helps to create test cases that are free of unnecessary details and redundant checks. Test cases themselves are authored using actions that make the tests readable and that can be automated without the need of programming skills.

Test Modules are best thought of as containers for organizing tests of user stories or software requirements. This way every test module will have a clear and well-differentiated scope from every other test module, which reduces redundant steps and checks, which makes tests less fragile and easier to maintain. Giving each Test Module a descriptive name makes it easy to identify what the test in the module cover.

### There are four sections to a Test Module: Objectives, Initial, Test Case and Final

The **OBJECTIVES** section lists every objective associated with the module’s test cases and defines the scope of the test module. Objectives allow readers to understand why test cases are designed the way they are, and give an auditor a quick insight into the correctness and completeness of a test.

The **INITIAL** section of a test module contains the action lines required for initialization of the test. For example, actions for launching the application under test would be written here, and would avoid repetitive steps in test cases when initialization is required multiple times.

The **TEST CASE** section is where test cases are created. Each test case has a representative number, accompanied by title and test objective line. The title and objective explain the test case purpose. Test Modules can have multiple test cases and each may have one or more test objectives associated with it.

The **FINAL** section is for any cleanup operation upon test completion, such as closing the application under test.

<table>
<thead>
<tr>
<th>TEST MODULE</th>
<th>ABT Test Module Template</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>OBJECTIVES</strong></td>
<td>ID</td>
</tr>
<tr>
<td>test objective</td>
<td>TO 1</td>
</tr>
<tr>
<td>test objective</td>
<td>TO 2</td>
</tr>
<tr>
<td>test objective</td>
<td>TO 3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>INITIAL</th>
<th>Setting up</th>
</tr>
</thead>
<tbody>
<tr>
<td>/set up steps: launch apps, setup test data, etc.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TEST CASE</th>
<th>TC 1</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>test objective</td>
<td>TO 1</td>
<td>Cause and effect</td>
</tr>
<tr>
<td>/steps and checks</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TEST CASE</th>
<th>TC 2</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>test objective</td>
<td>TO 2</td>
<td>Cause and effect</td>
</tr>
<tr>
<td>/steps and checks</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TEST CASE</th>
<th>Copy of Copy o...</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>test objective</td>
<td>TO 3</td>
<td>Cause and effect</td>
</tr>
<tr>
<td>/steps and checks</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FINAL</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>/clean up steps: delete created data, close browsers, close apps, etc.</td>
<td></td>
</tr>
</tbody>
</table>

See the full details for constructing an ABT test module at [www.testarchitect.logigear.com/action-based-testing/test-module-template.html](http://www.testarchitect.logigear.com/action-based-testing/test-module-template.html)
ABT and Keyword-driven testing - Similarities and Differences

Both Keyword-driven testing and Action Based Testing (ABT) use a test authoring approach that separates much of the programming work of test automation from the actual test design. This allows tests to be developed earlier and aids in tests maintenance.

Both methods use keywords symbolize test steps including interaction (e.g., UI, API, etc.) operations such as "click", "enter", and "select"; or business operations such as "Enter Transaction", "Delete Client", and so on.

<table>
<thead>
<tr>
<th>Keyword-driven testing</th>
<th>Action Based Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pros:</strong></td>
<td><strong>Pros:</strong></td>
</tr>
<tr>
<td>+ Is a free-form language for writing a tests</td>
<td>+ Is a top-down test development method</td>
</tr>
<tr>
<td>+ Is good for creating and documenting tests</td>
<td>+ Provides a structured language for writing tests</td>
</tr>
<tr>
<td>+ Can be used to write manual or automated tests</td>
<td>+ Makes tests self-documenting</td>
</tr>
<tr>
<td><strong>Cons:</strong></td>
<td>+ Makes tests fast to create</td>
</tr>
<tr>
<td>- Can create a test management issue.</td>
<td>+ Simplifies test maintenance</td>
</tr>
<tr>
<td>- Does not include test design or method</td>
<td>+ Can be used to create manual or automated tests</td>
</tr>
<tr>
<td></td>
<td>- Learning curve</td>
</tr>
</tbody>
</table>

Action Based Testing

ABT is a top-down test development method that consists of three parts:

1. Modular Test Planning
2. Test Module Development
3. Action-driven Test Case authoring and action design

From a syntax standpoint, actions and keywords are similar, however, ABT actions are dynamic; they can be modified and combined. Also ABT has two categories of actions:

- System-Level actions for interface interaction operations and other system-related and/or utility operations.
- Application-Level actions are often used for business operations.

Below is an example of an ABT test with 4 action lines. Column A has the action, and the subsequent columns, B, C, D, etc. have the arguments of the action. The action “rent car” (row 72) is consider as a business operation and “click” (row 75) and “check” (rows 78 and 79) are considered as UI operation action.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>71</td>
<td>rent car</td>
<td>country pickup</td>
<td>country return</td>
<td></td>
</tr>
<tr>
<td>72</td>
<td></td>
<td>Canada</td>
<td>Canada</td>
<td></td>
</tr>
<tr>
<td>73</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>74</td>
<td>click</td>
<td>window</td>
<td>control</td>
<td></td>
</tr>
<tr>
<td>75</td>
<td></td>
<td>welcome</td>
<td>new order</td>
<td></td>
</tr>
<tr>
<td>76</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>77</td>
<td>check</td>
<td>window</td>
<td>control</td>
<td></td>
</tr>
<tr>
<td>78</td>
<td></td>
<td>date and location</td>
<td>pickup country canada</td>
<td>on</td>
</tr>
<tr>
<td>79</td>
<td>check</td>
<td>date and location</td>
<td>return country canada</td>
<td>on</td>
</tr>
<tr>
<td>80</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Risk Based Testing—Strategies for Prioritizing Tests against Deadlines

When it is out of the question to delay delivery, the solution is a prioritization strategy in order to do the best possible job within the time constraints.

By Hans Schaefer

The bad game

You are in a bad game with a high probability of loosing: You will loose the game any way, by bad testing, or by requiring more time to test. After doing bad testing you will be the scape-goat for lack of quality. After reasonable testing you will be guilty for late release.

How to get out of the game?

You need some creative solution, namely you have to change the game. You need to inform management about the impossible task you have in such a way that they understand. You need to present alternatives. They need a product going out of the door, but they also need to understand the RISK.

One strategy is to find the right quality level. Not all products need to be free of defects. Not every function needs to work. Sometimes you have options about product quality. This means you can cut down testing in less important areas.

Another strategy is priority: Test should find the most important defects first. Most important means often "in the most important functions". These functions can be found by analyzing how every function supports the mission, and checking which functions are critical and which are not. You can also test more where you expect more defects. Finding the worst areas in the product soon and testing them more will help you find more defects. If you find too many serious problems, management will often be motivated to postpone the release or give you more time and resources. Most of this article is about a combination of most important and worst areas priority.

A third strategy is making testing faster in general. One major issue here is automation of test execution. But be cautious: Automation can be expensive, especially if you have never done it before or if you do it wrong! However, experienced companies are able to automate test execution with no overhead compared to manual testing.

A fourth strategy is getting someone else to pay. Typically, this someone else is the customer. You release a lousy product and the customer finds the defects for you. Many companies have applied this. For the customer this game is horrible, as he has
no alternative. But it remains to be discussed if this is a good strategy for long term success. So this "someone else" should be the developers, not the testers. You may require the product to fulfill certain entry criteria before you test. Entry criteria can include certain reviews having been done, a minimum level of test coverage in unit testing, and a certain level of reliability. The problem is: you need to have high-level support in order to be able to enforce this. Entry criteria tend to be skipped if the project gets under pressure and organizational maturity is low.

The last strategy is prevention, but that only pays off in the next project, when you, as the test manager, are involved from the project start on.

1. Understanding necessary quality levels

Software is embedded in the larger, more complex business world. Quality must be considered in that context.

You have to be sure which qualities and functions are important. Fewer defects do not always mean more profit! You have to research how quality and financial performance interact. Examples of such approaches include the concept of Return on Quality (ROQ) used in corporations such as AT&T. ROQ evaluates prospective quality improvements against their ability to also improve financial performance. Be also aware of approaches like Value Based Management. Avoid to fanatically pursuing quality for its own sake.

Thus, more testing is not always needed to ensure product success!

2. Priority in testing most important and worst parts of the product

Risk is the probability for damage to occur. The way to assess risk is outlined in figure 1. Risk analysis assesses damage during use, usage frequency, and determines probability of failure by looking at defect introduction.

Testing is always a sample. You can never test everything, and you can always find more to test. Thus you will always need to make decisions about what to test and what not to test, what to do more or less. The general goal is to find the worst defects first, the ones that NEED TO BE FIXED BEFORE RELEASE, and to find as many such defects as possible.

This means the defects must be important. The problem with most systematic test methods, like white box testing, or black box methods like equivalence partitioning, boundary value analysis or cause-effect graphing, is that they generate too many test cases, some of which are less important. A way to lessen the test load is finding the most important functional areas and product properties. Finding as many defects as possible can be improved by testing more in bad areas of the product. This means you need to know where to expect more defects.

When dealing with all the factors we look at, the result will always be a list of functions and properties with an associated importance. In order to make the final analysis as easy as possible, we express all the factors in a scale from 1 to 5. Five points are given for "most important" or "worst", or generally for something having higher risk, which we want to test more, 1 points is given to less important areas.

3.1. Determining damage: What is important?

You need to know the possible damage resulting from an area to be tested. This means analyzing the most important areas of the product. In every product, there may be other factors playing a role, but the factors given here have been valuable in several projects.

Important areas can either be functions or functional groups, or properties such as performance, capacity, security etc. The result of this analysis is a list of functions and properties or combination of both that need attention. I am concentrating here on sorting functions into more or less important areas. The approach, however, is flexible and can accommodate other items.

You have to analyze the use of the software within its overall environment. Analyze the ways the software may fail. Find the possible consequences of such failure modes, or at least the worst ones. Take into account redundancy, backup facilities and possible manual check of software output by users, operators or analysts. Software that is directly coupled to a process it controls is more critical than software whose output is manually reviewed before use. If software controls a process, this pro-
cess itself should be analyzed. The inertia and stability of the process itself may make certain failures less interesting.

Potential damage may be classified into the classes mentioned below, or quantified into money value, whichever seems better. In systems with large variation of damage it is better to use damage as absolute money value, and not classify it into groups.

A failure would be catastrophic (3)

Such failures may deal with large financial losses or even damage to human life. An example would be the gross uncoupling of all subscribers to the telephone network on a special date.

Failures leading to loosing the license, i.e. authorities closing down the business, are part of this class. Serious legal consequences may also belong here.

The last kind of catastrophic failures is endangering the life of people.

A failure would be damaging (2)

The program may not stop, but data may be lost or corrupted, or functionality may be lost until the program or computer is restarted.

A failure would be hindering (1)

The user is forced to workarounds, to more difficult actions to reach the same results.

A failure would be annoying (0)

This issue would be a problem that doesn’t affect functionality, but rather it makes the product less appealing to the user or customer. The significant risk is while the customer can live with the problem there is possible loss of faith by customers, which would mean longer-term loss of business because customers may avoid future purchase of products from the company.

Damage can also depend on how often a function or feature is used. Some functions may be used every day, other functions only a few times. Some functions may be used by many, some by few users. Give priority to the functions used often and heavily. The number of transactions per day may be an idea helping in finding priorities.

Sometimes this analysis is not quite obvious. In process control systems, for example, certain functionality may be invisible from the outside. In modern object oriented systems, there may be a lot of central libraries used everywhere. It may be helpful to analyze the design of the complete system.

Following is a hierarchy of functionality by user interaction:

Unavoidable (3)
An area of the product that most users will come in contact with during an average usage session (e.g. startups, printing, saving).

Frequent (2)
An area of the product that most users will come in contact with eventually, but maybe not during every usage session.

Occasional (1)
An area of the product that an average user may never visit, but that deals with functions a more serious or experienced user will need occasionally.

Rare (0)
An area of the product which most users never will visit, which is visited only if users do very uncommon steps of action. Critical failures, however, are still of interest.

Importance can be classified by using a scale from one to five. However, in some cases this does not sufficiently map the variation of the scale in reality. Then, it is better to use real values, like the cost of damage and the actual usage frequency.

### Determining probability of failure

![Probability of failure diagram]

\[
\text{Probability} = \frac{\text{defect density}}{\text{volume}}
\]

### 3.2. Failure probability: What is (presumably) worst?

The worst areas are the ones having most defects. The task is to predict where most defects are located. This is done by analyzing probable defect generators. In this section, some of the most important defect generators and symptoms for defect prone areas are presented. There exist many more, and you have to always include local factors in addition to the ones mentioned here.

- Complex areas
  Complexity is maybe the most important defect generator. More than 200 different complexity measures exist, and research into the relation of complexity and defect frequency has been done for more than 20 years.
• Changed areas
Change is an important defect generator. One reason is that changes are subjectively understood as easy, and thus not analyzed thoroughly for their impact. Another reason is that changes are done under time pressure and analysis is not completely done.

• Impact of new technology, solutions, methods
Programmers using new tools, methods and technology experience a learning curve. In the beginning, they may generate many more faults than later. A good example is the first project with a new type of user interface. The general functionality may work well, but the user interface subsystem may be full of trouble.

• Impact of methods
Another factor to consider is the maturity of methods and models. Maturity means the strength of the theoretical basis or the empirical evidence. If software uses established methods, like finite state machines, grammars, relational data models, and the problem to be solved may be expressed suitably by such models, the software can be expected to be quite reliable. On the other hand, if methods or models of a new and unproven kind, or near the state of the art are used, the software may be more unreliable.

Most software cost models include factors accommodating the experience of programmers with the methods, tools and technology. This is as important in test planning, as it is in cost estimation.

• Impact of the number of people involved
The idea here is the thousand monkeys’ syndrome. The more people are involved in a task, the larger is the overhead for communication and the chance that things go wrong. A small group of highly skilled staff is much more productive than a large group of average qualification. In the COCOMO software cost model, this is the largest factor after software size. Much of its impact can be explained from effort going into detecting and fixing defects.

Areas where relatively many and less qualified people have been employed, may be pointed out for better testing.

Care should be taken in that analysis: Some companies employ their best people in more complex areas, and less qualified people in easy areas. Then, defect density may not reflect the number of people or their qualification.

• Impact of turnover
If people quit the job, new people have to learn the design constraints before they are able to continue that job. As not everything may be documented, some constraints may be hidden for the new person, and defects result. Overlap between people may also be less than desirable.

In general, areas with turnover will experience more defects than areas where the same group of people has done the whole job.

• Impact of time pressure
Time pressure leads to people making short-cuts. People concentrate on getting the job done, and they often try to skip quality control activities, thinking optimistically that everything will go fine. Only in mature organizations does this optimism seems to be controlled.

Time pressure may also lead to overtime work. It is well known, however, that people loose concentration after prolonged periods of work. This may lead to more defects. Together with short-cuts in applying reviews and inspections, this may lead to extreme levels of defects density.

• Areas with many defects before
Defect repair leads to changes which lead to new defects, and defect prone areas tend to persist. Experience exists that defect prone areas in a delivered system can be traced back to defect prone areas in reviews and subsystem testing. Evidence in studies and shows that modules that had faults in the past are likely to have faults in the future. If defect statistics from design and code reviews, and unit and subsystem testing exist, then priorities can be chosen for later test phases.

• Communication issues
If people working together on a project are not co-located, communication will be worse. This is true even on a local level.

In principle, geographical distribution is not dangerous. The danger arises if people with a large distance have to communicate, for example, if they work with a common part of the system. You have to look for areas where the software structure implies the need for good communication between people, but where these people have geography against them.
• One factor to be considered in general
  This article is about high level testing. Developers test before this. It is reasonable to have a look at how developers have tested the software before, and what kind of problems they typically overlook. Analyze the unit test quality. This may lead to a further tailoring of the test case selection methods.

Looking at these factors will determine the fault density of the areas to be tested. However, using only this will normally over-value some areas. Typically, larger components will be tested too much. Thus, a correction factor should be applied: Functional size of the area to be tested. I.e. the total weight of this area will be "fault proneness / functional volume". This factor can be found from function point analysis early, or from counting code lines if that is available.

What to do if you do not know anything about the project, if all the defect generators can not be applied?

You have to run a test. A first rough test should find defect prone areas, the next test will then concentrate on them. The first test should cover the whole system, but be very shallow. It should only cover typical business scenarios and a few important failure situations, but cover all of the system. You can then find where there was most trouble, and give priority to these areas in the next round of testing. The next round will then do deep and thorough testing of prioritized areas.

3.3. How to calculate priority of test areas

The general method is to assign weights, and to calculate a weighted sum for every area of the system. Test where the result is highest!

For every factor chosen, assign a relative weight. You can do this in very elaborate ways, but this will take a lot of time. Most often, three weights are good enough. Values may be 1, 3, and 10. (1 for "factor is not very important", 3 for "factor has normal influence", 10 for "factor that has very strong influence").

For every factor chosen, you assign a number of points to every product requirement (every function, functional area, or quality characteristic. The more important the requirement is, or the more alarming a defect generator seems to be for the area, the more points. A scale from 1 to 3 or 5 is normally good enough. Assigning the points is done intuitively.

The number of points for a factor is then multiplied by its weight. This gives a weighted number of points between 1 and 50. These weighted numbers are then summed up for damage (impact) and for probability of errors, and finally multiplied. As many intuitive mappings from reality for points seem to involve a logarithmic scale, where points follow about a multiplier of 10, the associated risk calculation should ADD the calculated weighted sums for probability and damage. If most factors' points inherently follow a linear scale, the risk calculation should MULTIPLY the probability and damage points. The user of this method should check how they use the method! Testing can then be planned by assigning most tests to the areas with the highest number of points.

### Table: Test Area Prioritization

<table>
<thead>
<tr>
<th>Area to test</th>
<th>Business criticality</th>
<th>Visibility</th>
<th>Complexity</th>
<th>Change frequency</th>
<th>RISK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weight</td>
<td>3</td>
<td>10</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>Order registration</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>1</td>
<td>46*18</td>
</tr>
<tr>
<td>Invoicing</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>2</td>
<td>62*18</td>
</tr>
<tr>
<td>Order statistics</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>16*18</td>
</tr>
<tr>
<td>Management reporting</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>16*18</td>
</tr>
<tr>
<td>Performance of order registration</td>
<td>5</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>55*6</td>
</tr>
<tr>
<td>Performance of statistics</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>13*6</td>
</tr>
<tr>
<td>Performance of invoicing</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>22*6</td>
</tr>
</tbody>
</table>

**Note:** The table reflects an organized approach to prioritizing test areas based on the factors of business criticality, visibility, complexity, and change frequency. The calculation of risk involves multiplication of these factors, with some weighing more than others. The table serves as a guideline for prioritization, where areas are ranked based on their calculated RISK values.

In Figure 2, the table suggests that function «invoicing» is most important to test, «order registration» and performance of order registration. The factor which has been chosen as the most important is visibility.

A word of caution: The assignment of points is intuitive and may be wrong. Thus, the number of points can only be a rough guideline. It should be good enough to distinguish the high-risk areas from the medium and low risk areas. That is its main task. This also means you don't need to be more precise than needed for just this purpose. If more precise test prioritization is necessary, a more quantified approach should be used wherever possible.
4. Making testing more effective

More effective test means to find more and more important defects in the same amount of time.

The strategy to achieve this is to learn from experience and adapt testing.

First, the whole test should be broken into four phases:

- test preparation
- pre-test
- main test
- after-test.

Test preparation sets up areas to test, the test cases, test programs, databases and the whole test environment.

Especially setting up the test environment can give a lot of trouble and delay. It is generally easy to install the program itself and the correct operating system and database system. Problems often occur with the middleware, i.e. the connection between software running on a client, and software running on different servers. Care should be taken to thoroughly specify all aspects of the test environment, and dry runs should be held, in order to ensure that the test can be run when it is time to do it.

The pre-test is run after the software under test is installed in the test lab. This test contains just a few test cases running typical day to day usage scenarios. The goal is to test if the software is ready for testing at all, or totally unreliable or incompletely installed. Another goal may be to find some initial quality data, i.e. find some defect prone areas to focus the further test on.

The main test consists of all the pre-planned test cases. They are run, failures are recorded, defects found and repaired, and new installations of the software made in the test lab. Every new installation may include a new pre-test. The main test takes most of the time during a test execution project.

The after-test starts with every new release of the software. This is the phase where optimization should occur. Part of the after-test is regression testing, in order to find possible side-effects of defect repair. But the main part is a shift of focus.

The type of defects may be analyzed. In principle, every defect is a symptom of a weakness of some designer, and it should be used to actively search for more defects of the same kind.

5. Strategies for prevention

The starting scenario for this article was the situation where everything is late and where no professional budgeting has been done. In most organization, there exist no experience data and there exists no serious attempt to really estimate costs for development, testing, and error cost in maintenance. Without experience data there is no way to argue about the costs of reducing a test.

The imperatives are:

- You need a cost accounting scheme
- You need to apply cost estimation based on experience and models

You need to know how test quality and maintenance trouble interact

Measure:

- Size of project in lines of code, function points etc.
- Percentage of work used in management, development, reviews, test preparation, test execution, and rework
- Amount of rework during first three or six months after release
- Fault distribution, especially causes of user detected problems.
- Argue for testing resources by weighting possible reductions in rework before and after delivery against added testing cost.

Summary

Testing in a situation where management cuts both budget and time is a bad game. You have to endure and survive this game and turn it into a success. The general methodology for this situation is not to test everything a little, but to concentrate on high risk areas and the worst areas.

Priority 1: Return the product as fast as possible to the developers with a list of as serious deficiencies as possible.

Priority 2: Make sure that, whenever you stop testing, you have done the best testing in the time available!

(This is a shortened version of the article originally published on http://www.methodsandtools.com/archive/archive.php?id=31)

Hans Schaefer has been running his own company since 1987, specializing in consulting about software testing and software quality matters. He is teaching seminars about software testing, mostly in Scandinavian countries, Germany and China.

He has worked for most leading Norwegian companies, as well as companies like Bombardier, DNB, Ericsson, Nokia, Norwegian Tax Authorities, Statoil, Siemens, Telenor, University of Iceland, Visma.
For many, perhaps most, development teams the terms requirement and specification are used interchangeably with no detrimental effect. In everyday development conversations the terms are used synonymously, one is as likely to mean the "spec" as the "requirements."

However it is sometimes useful, and occasionally important, to differentiate between the two terms:
"A requirement is a desired relationship among phenomena of the environment of a system, to be brought about by the hardware/software that will be constructed and installed in the environment. A specification describes machine behaviour sufficient to achieve the requirement. A specification is a restricted kind of requirement."

The key points to note are in the last two sentences:
- A specification describes behaviour to achieve requirement.
- Specification is a restricted requirement, i.e. the specification narrows down the requirement.

For example: there may be a requirement to store customers details for shipping and future marketing. The specification would state what details should be stored (e.g. name, postal address, e-mail address, etc.). Specifications can be very detailed, e.g. a postal address should contain an house number or name, a street, a post code, and the format the post code should satisfy.

**Specifications**

In creating the specification the requirement may change. For example: should the system accept US style zip codes as well as UK style postal code? This depends on whether the system is required to service only UK customers. Consequently those commissioning the system might need to consider their international approach.

- In exposing the detail of the specification the requirement may be brought into question, refined and even changed. A question of detail may ripple all the way up to the strategic level. Although, as with good code, one hopes that such ripples will not occur that often.
- If questions arising from specifications regularly ripple back, it may be a sign that the requirements, encapsulations or even strategy and goals are weak.

- There is almost no limit to the detail a specification can reach. In University I was taught to write incredibly detailed specifications in formal, mathematical, logical notation called VDM-SL (Jones, 1986). Yet for many teams this level of detail is unnecessary and for most teams is not economically justified.

**Specifications In creating the specification the requirement may change.**

For many teams the specifications are uncovered as part of the coding process. Indeed code itself represents the ultimate specification of what happens. Unfortunately in this form the specification is difficult for non-programmers to understand and therefore agree to and, more importantly, verify.

In some fields leaving the specification to the programmers is a good thing. Programmers who understand the field may have little need of additional (expensive) documentation; in fast changing environments writing down a specification and communicating may injected undesirable delays.

In other fields it is preferable to have the specification understood in advance or determined by specialists.

By Allan Kelly

**Something Old, Something New: Requirements and Specifications**

Differences in interpretation of requirements and specifications by programmers and testers is a common source of bugs.
organizations may agree on specifications to allow completing products to interoperate. For example, passing short SMS text message between competing handsets over competing networks using competing switching equipment requires all parties to follow agreed specifications.

For teams working in a traditional - upfront requirements, specification and design - specifications can become a battleground. Programmers put under pressure, without knowledge or specifications, inevitably do things the consuming clients do not expect. One side or other will demand more detail next time to prevent the problem.

But more detail doesn't solve the problem because a) nobody remembers the details, b) omissions and mistakes are made in specifying the detail, c) more detail leads to more things that can change, more things to be read (and forgotten) and more opportunities for mistakes in the detail.

Since the amount of detail is almost infinite the call "for more detail" easily escalates into an arms-race. Introducing more detail in specifications can quickly make things worse not better.

**For example**

Consider the CEO of a large super market chain. His strategy is for increased market share. He, and his board, is prepared to trade profits for market share. The requirement he gives his COO is: increase sales.

Given this requirement the COO convenes his team. They determine that some B2G1F offers - Buy two get one free - are called for. They task the marketing team with deciding which products to apply the offers to and the IT team with providing the systems to implement this.

The IT team receives the B2G1F requirement and quickly realize that one requirement is for the products it applies to must be configurable. But how configurable? Does the marketing team require a web interface? Or can it be managed through a XML config file? The original requirement expands into multiple small requirements.

Then there is a question of how the offer is implemented. Requirements become very specific. Obviously when a customer buys two identical products a third identical product is free. But what if the customer buys two of the products in large and a third small one? Is the third small? Or one of the large?

And when the marketing team says "1 litre fruit juices" does that mean that someone buying 1 litre each of orange juice, apple juice and cranberry juice gets one free? And does that mean the B2G1F offer needs to be marketed differently? Requirements can be large, they can be small, they can hide details which later become significant. Over time requirements are refined, they are atomised and details added. At some point a requirement becomes a specification.

**Enter the Iteration**

When working in short iterations requirements are best given at the start of each iteration - not all requirements need to be known in advance but enough for the duration of the iteration should be.

Teams which embrace unplanned work can happily start an iteration with missing requirements or respond to unexpected requirements. Teams which aim for maximum predictability will see unplanned work as problematic and aim to pin requirements down in advance.

Specifications on the other hand might be known in advance or might be discovered during the iteration, either as a specific exercise or as part of the coding activity. Sometime leaving programmers to finalise specification is not only possible but advantageous. Other times specifications might be determined in advance by a specialist, typically an analyst of some description.

**Figure 1 - Requirements go in, working software comes out**

Problems occur when specification are decided far in advance. When this is done specifications decay because:

- A changing world leads to specifications and requirements that change too.
- As the development team creates the system they learn about both problem and solution domain, this can lead them to new insights.
- Without a deadline analysis can continue almost indefinitely, allowing more time for work to occur most likely leads to more work.
- The more specifications are decided the more that can change.
- Pre-emptively creating specifications will increase the amount of work done which is later discarded. Delaying specification creating reduces the chances that development work is cancelled, removed or changes after specifications are created.
One source puts the rate of requirements and specification change at 2% per month (Jones, 2008), which works out at an annual compound rate close to 27%. Changed requirements means changed specifications which leads to rework.

Therefore it is preferable to decide specifications as late in the day as possible - say in the same sprint as coding will occur, or in the previous sprint. In the early descriptions of Behaviour Driven Development (BDD) Dan North described a Business Analyst working at the same keyboard as a programmer writing specifications as code was written.

And tests

A development story, especially when in User Story format (Cohn, 2004), is usually a requirement. It is a token for work to be done and is often called a placeholder for a conversation. The acceptance criteria often found on the back of a story card are specification but rarely are they a complete specification. Detail can be pinned down later in a conversation.

In some teams a fuller specification will be created in the form of acceptance criteria produced by a requirements engineer or professional tester before coding begins. If this is not done then the specification will be completed at the time of coding or at the time of testing.

Differences in interpretation of requirements and specification by programmers and testers is a common source of bugs. Two individuals read the same document, the programmer interprets it one way and writes code as such; the professional tester interprets it differently and tests it as such.

Formal methods removed this problem by using exact logical notation but in doing so make the specifications difficult for novices to read and increase the chances of errors in the specification itself. Another solution is to use acceptance test scripts derived from acceptance criteria as the most detail form of specification. When these are written in natural language there is room for ambiguity. When written in a formal form ambiguity is squeezed from the system. When the formal form is executable - such as a FIT table or Gerkin given when then then it is possible to ensure the program code and specification match. This is an executable specification.

Whether in a logic based notation such as VDM-SL or pseudo-English Gerkin the aim is the same: a specification that can be executed by a machine. The difference is when this execution occurs:

- Gerkin style executable specifications are used as after code is written as tests to ensure the human programmer produces the right thing.
- Logic based formal specifications aim to direct the code itself, either by executing the logic directly or by machine translation to a language such as C.

Although ambiguity may be squeezed out of specifications by formalizing them it is more difficult to eliminate omissions. To extend the earlier example, tests may be used to ensure an address post-code matches the prescribed format but tests cannot ensure customers supply their county unless a human intervenes to specify county as a necessary field.

How much detail specifications and tests need to specify, and the point at which the details are decided varies greatly. For some teams specification can be left in the hands of the programmer when they are coding. In other environments specifications needed to be pinned down by specialists well in advance.

Whenever specifications and tests are to be used they should be created before coding begins. Too create them after the programmer has completed their work is to invite discrepancies and rework. While this is not guaranteed to eliminate problems it can significantly reduce them.

Automated acceptance tests: the new formal methods

Automated acceptance tests, also known as executable specifications, continue the tradition of formal logic specifications. The tests are a specification. Automation demands formalization because automation requires code. The first difference is that Gerkin style given-when-then specifications are readable by most people while programming logic is only meaningful to those with years of experience.

It is interesting to note that the given-when-then specification format mirrors the pre-post conditions used in formal languages. The given declares a set of pre-conditions and the then declares the post-conditions.

Secondly both techniques require tool support to be effective. But while predicate logic specifications tools are few and far between, expensive and difficult to use the tools used for executable specification are largely available free of charge as open source, e.g. Selenium, JBehave, Cucumber and FIT/FITNESSE.
Rather than providing a logical description of the program under development these tools work through examples. Specifications are given by way of examples - hence the name specification by example. Because these examples are executable as tests it is possible to validate the program satisfies these specifications.

These examples may not be exhaustive, there may be undefined behaviour in the system where specification examples are not provided. The program code is still the ultimate specification, the examples aim to cover observable behaviour where it is of significance to those doing the specification. (This differs from predicate logic descriptions which aim to be complete).

Because BDD and ATDD tests are machine executable they are actually a form of formal specification. This author believes a useful line of research would be to see if such tests could be transformed in more traditional logical notations like VDM and Z.

Knowledge and Trust

Whether specifications are needed or not often boils down to knowledge and trust. When developers have extensive knowledge of the domain they are working in then much of the information that would be contained in specifications already exists inside their heads. And when they have ready access to others who know more about the domain a verbal conversation may substitute for a written document.

For example, while most programmers will be familiar with the postal address example given previously, only those who work in specialist domains will know other formats. English legal practitioners frequently use DX mail rather than Royal Mail. Programmers who have worked in legal software for some years may automatically provide DX number and exchange in software while those new to the field need to be told, i.e. they need to be provided with a specification, this may be written or verbal.

However knowledge alone may not be enough if programmers and testers are not trusted to use their own knowledge, or not able to ask for assistance when they recognise they do not. Forgoing specification documents saves money because documentation is expensive. It also accelerates development because writing documentation is a time consuming process prone to blocking. Having a programmer determine specification as they code is the ultimate in just-in-time working.

Still there may be merit in having another person bring their knowledge and understanding to the specification effort. If these specifications are to match the code they must be created in some fashion which minimise opportunities for differences in understanding to emerge.

On anything other than trivial systems using human effort to validate that specifications and program match becomes a time consuming and error prone exercise, and thus slow and financially expensive. To overcome this specifications need to be both machine readable and executable - through automated tests or theorem validation - to ensure code and test specification say the same thing.

When knowledge and trust are lacking specifications become necessary, so too does an effective, usually automated, means of validating code against specification.

30 years ago the software industry attempted to solve the specification problem with formal methods. The BDD and ATDD techniques in use today take a similar approach but with a far lower barrier to entry. They in effect reinvent formal specification.

Conclusion

- There is often little point in differentiating between Requirements and Specification and the two terms are often used to mean the same thing, i.e. the thing to be built.
- Distinguishing between specifications and requirements can add to understanding.
- Requirements are best given at the start of each iteration but specifications can be discovered within the iteration. Finalising specifications as late as possible has a number of advantages.
- Requirements are unavoidably imprecise. Specifications should not be.
- Discovering specifications can lead to changes in the requirements. Requirements come before specifications but specifications can send ripples back to requirements.
- Specifications are test criteria; both specifications and test criteria can be formalised. Formalising specifications as predicate logic is time consuming and rarely justified. Formalising tests as executable specifications can be highly effective.

(This article was originally published on http://www.methodsandtools.com/archive/requirementspecification.php)

Allan Kelly has held just about every job in the software world, from system admin to development manager by way of programmer and product manager. Allan works for Software Strategy providing consulting and training to help teams adopt and deepen agile practices. He specialises in working with software product companies and aligning products and processes with company strategy.

Allan is also the author of two complete books: "Business Patterns for Software Developers" and "Changing Software Development: Learning to be Agile", and one work in progress "Xanpan – team centric Agile software development". He is also the originator of Retrospective Dialogue Sheets.

More about Allan at http://www.allankelly.net and on Twitter as @allankellynet (http://twitter.com/allankellynet).
5 Tips for Getting Software Testing Done in the Scrum Sprint

A clear practice with the support of tools will help teams be more successful in delivering done products.

By Clemens Reijnen

It is always challenging to create a piece of a software system that fulfills a customer need, ready for use. Especially when it should be realized in an iteration of just three weeks, from an idea to a fully functional and tested piece of the application. Achieving the 'Definition of Done' as it is called in the Scrum approach.

Agile approaches embrace short iteration cycles where releasable pieces of the software system are created. Releasable means also tested. Unit tested, system tested, functional tested, acceptance tested and often also performance and load tested. Making the item ready for use should allow providing feedback for the development team. Feedback how they are doing and if it is what the customer really needed.

Many teams find this a troublesome challenge and aren't successful in it. They deliver half-done products or a make workaround for the agile approach (in Scrum often called 'Scrum but'). Many agile workarounds contradict the goal why a team adopted an agile approach. For example a standalone test team or a separate test iteration to get the testing done will result in less agility and less feedback loops.

This article will give you five tips how a clear practice with the support of tools will help teams be more successful in delivering done products when using an agile approach.

Actually many tips will also be helpful for other methodologies and project approaches. This article uses Microsoft Application Lifecycle Management tools as an example, but the tips are valid for any other ALM tool suite.

Many readers will possibly think "tools, wasn't that evil in agile? People, interactions versus tools and processes". This is half-correct: it isn't evil and yes interactions are very important and solve miscommunications way better than tools and processes ever can. However, tools can help. Tools and practices can support a way of working. Application Lifecycle Management tools suite, integrated tools with a central repository for all involved roles support collaboration between roles. They support collaboration between artifacts these roles create and teamwork between the work these roles execute. A recent Gartner report says that "Driven by cloud and agile technologies, the ALM market is evolving and expanding."

Tip 1: Get a team

This is actually not a tip, it is a must. This is a kind of obvious but not common and the hardest thing to accomplish. Get a team, get testing knowledge in your team. When you don't have it, you will fail. Teams and companies have failed to reach their agile software development goals only because it was impossible to get different disciplines together in a team.

For example, the code implementation is done in an agile way, with Scrum boards and daily standups together with the customer. This is done because the customer wanted to be more flexible in what is needed in the system. However, software testing is done in a separate iteration and cadence because this role is the responsibility of a different department. Bugs are found in functionality realized sprints ago, testers needs more detailed requirements descriptions because they don't understand the backlog items, pushing the customer in the corner to be more descriptive and fixed till testing was done. The customer loses all the flexibility he needed and gets frustrated.

This is just a simple example how it could go wrong when you don't have a team. And there are thousands more. It isn't easy to accomplish a collaborative environment where all roles work seamless together. Testers and developers are different, as a nice quote from this test blog [2] describes it:
In the D-world, the world of the Developers, we think Generalist Testers are pencil-pushing, nit-picky quality geeks. Mostly they are beside the point and are easily replaced. They seem to like making much noise about little defects, as if we made those errors deliberately....

In the T-world we don't hate the Developers for their perceptions. We are disappointed about the poor quality of the software. Bad assumptions on the part of Developers are more to blame for the problems than are software weaknesses.

We never (or seldom) get software what will work right the first time. No, in the T-world we think that developers forget for whom they are building software, it looks like they are building for themselves...

If you try to combine these two worlds in one team, you definitely need to come up with a Collaborative Culture

The three most important concerns are:
- Trust.
- A topic closely associated with trust when it refers to people is Identity.
- Collaborative culture.
- A collaborative culture consists of many things, including:
  - Collaborative leadership;
  - Shared goals;
  - Shared model of the truth; and
  - Rules or norms.
- Reward.
- A "reward" for successful collaboration is most often of a non-financial nature.

Show me the value, seems to be the magic word. Test adds knowledge, knowledge during the grooming of the backlog. They help the product owner with defining proper acceptance criteria. Testers can help find improper written backlog items, finding inconsistencies in the flow of a business case for example. A great test activity in the TMap testing approach can help, assessing the test base. TMap is a test management approach which structures the testing effort by providing different phases and tasks, see the TMap.NET web site for more details. Simply explained: tests find bugs in the requirement. In both ways the test role helps the product owner and the team to focus on value.

Tools can help. The Visual Studio 2010 TMap Testing Process template gives test activities a more important place, helping the tester to get on board.

Visual Studio Process Templates are supporting a way of working. They contain several work items types with a flow.

For example, a bug work item type can go from the status 'new' to 'assigned' to 'resolved' and 'verified'. Such a work item can hold a lot of information, supporting the work that needs to be done to bring the work item to the next status. A process template is easy to customize and work item type fields, flow, validation and rights can be edited. Creating a new type is also supported. For example the TMap Testing Process Template has an additional type "Test Base Finding", helping the management of problems found in the test base (backlog).

The 'Testing' Tab with test activities, next to the implementation tab.

Still two different worlds in this way, but it gives a good visual reward of being connected. Many teams probably won't need an additional visualization of the testing effort and can use the Scrum process template in combination with their testing methodology. This will help them to get started.

The manual test tool 'Microsoft Test Manager' (MTM) in Visual Studio is interesting. It helps teams to get more connected as it shows the pain points where the collaboration isn't seamless. Adopting MTM can thus be a good start for agile teams to get testing aboard, but be aware the interactions are more important as tools. The tools won't fix bad collaboration, mismatching identities, lack of trust and won't give any reward.

**Tip 2: Write logical acceptance tests**

The previous tip "get a team" already explained the benefit of having testing knowledge onboard during the requirements gathering. Two practices are mentioned: assessing the test base and helping with acceptance criteria. This tip is close to the second practice, when the team benefits from being able to capture acceptance criteria in logical test cases.
During the release planning meeting, capture acceptance criteria and immediately add them as logical test cases linked to the product backlog item. This will help the team to understand the item and clarify the discussion. An even more important benefit of this tip is that it helps testers be involved and be important at the early stages of the software cycle.

With product backlog items you could use the following use story style for the description (or title): -- As a [role] I want [feature] So that [benefit] --

You can use a similar format for acceptance criteria: -- Given [context] And [some more context] When [event] Then [outcome] And [another outcome] --

Acceptance criteria are written like a scenario. SpecFlow (see SpecFlow website) is a Behavior Driven Development tool that also uses this way of describing scenario's, from where it binds the implementation code to the business specifications.

Tools can help to immediately create and link test cases to a backlog item. Having them present for further clarification and ready to get physical specified with test steps. Visual Studio Process templates support this kind of scenario.

A Product Backlog Item can have the fields 'description' and 'acceptance criteria' (see image).

Once I tried to implement this practice in a project the testers didn't agree. They were afraid the developers only would implement the functionality that was written in the logical test cases. For them, knowing on beforehand what was going to be tested seemed a bad idea for them. I had to work on Tip 1 first before the team could move forward.

**Tip 3: Use a risk and business driven test approach**

When there is no risk, there is no reason to test. So, when there isn't any business risk, there aren't any tests and is it easy to fit testing in a sprint. More realistically, a good risk analysis on your product backlog items before starting to write thousands of test cases is a healthy practice.

Risk is also an important attribute in Scrum. The release plan establishes the goal of the release, the highest priority Product Backlog, the major risks, and the overall features and functionality that the release will contain.

Products are built iteratively using Scrum, wherein each Sprint creates an increment of the product, starting with the most valuable and riskiest.

Product Backlog items have the attributes of a description, priority, and estimate. Risk, value, and necessity drive priority. There are many techniques for assessing these attributes.

Product risk analysis is an important technique within the TMap test approach. Risk analysis is part of the proposed activities in the Master Test Plan of TMap: 'Analyzing the product risks'. It not only helps the Product Owner to make the right decisions, but it also
gives the team advantage in a later stage. Risk classification is invaluable while defining the right test case design techniques for the Product Backlog Item.

“The focus in product risk analysis is on the product risks, i.e. what is the risk to the organization if the product does not have the expected quality?”

Having a full product risk analysis for every Product Backlog Item during the Release Planning meeting is slightly overdone, but the major risks should be found. Determining the product risks at this stage will also provide input for the Definition of Done list.

Within the Visual Studio Scrum 1.0 Process Template Product Backlog Items are written down in the Work Item Type ‘Product Backlog Item’. This Work Item Type hasn’t got a specific field for risk classifications. Adding a risk field is done easily. To integrate testing in a sprint, you should know the risks and use test design techniques that cover those risks, writing only useful test cases.

Tip 4: Regression Test Sets

In the same context as tip 3 you can think of regression tests sets. Some teams rerun every test every sprint, this is time consuming and isn’t worth the effort. Having a clear understanding of what tests to execute during regression testing raises the return of investment of the testing effort and gives more time to specify and execute test cases for the functionalities implemented during the current sprint.

Collecting a good regression set is important. There are a lot of approaches how to get this regressions set, most of them are based on risk classifications and business value (see the previous tip).

The principle is that from each test case a collection of additional data is determined into the test cases for the regression test are 'classified'. Using these classifications all cross-sections along the subsets of test cases can form the total tests that are selected.

Automation of this regression set is almost a must (see next tip: test automation). Making a good selection which test cases to select is a trivial task. With excel you can do some querying for proper test cases but this gets harder when they are in different documents. Testing is more efficient if you have good query tools so you can easily make a selection (and change this selection) of the test cases are part of the regression run.

A team I supported had more than 15,000 test cases distributed over about 25 feature test plans and 10 Scrum teams. For the execution of a the regression set, a query needed to be run over all test cases to create a meaningful selection for the regression set.

Test cases in Team Foundation Server are stored as work Item Types in the central database brings, which has powerful query capabilities. You can write any query you want, save it and use it for your regression test selection. The team I supported used query based test suites to save the selections.

Microsoft Test Manager has an interesting capability to control the amount of regression testing that need to be done during the sprint. A feature called “Test Impact”, gives information about test cases which are impacted by code changes.

Tip 5: Test Automation

All validation activities (test) cost time and money. So, every activity to test a system should be executed as efficiently as possible (see previous tips). Adding automation to the execution of these validations saves execution time, which saves money. But the creation and especially the maintenance of test automation cost time and money. So, the hard question is “what and how should we automate for our system validations”, where is the breakeven point of test automation in the project. The ROI of test automation is a challenge. We have to
think of **how long** is the test automation relevant in our project (for example not all functional tests aren’t executed every sprint, only a sub set, only the most important, see this post ['only meaningful tests']) and **how many times** is the validation executed (how many times over time and also on different environments). This gives us indications on how much effort we must put in our test automation.

There are three basic test automation levels:
- No Automation
- Test Case Record and Playback
- Test Scripts

Visual Studio adds two other levels.
- No Automation
- Shared steps with Record and Playback (action recording)
- Test Case Record and Playback (action recording)
- Test Scripts (Generated Coded UI)
- Test Scripts (Manual created Coded UI)

Any other test automation tool will probably add his own value, but let’s focus on Visual Studio levels.

All these automation levels have an investment, and a maintainability level. The better you can maintain a test case, the longer you can use it for your ever evolving software system. That is the connection between ‘how long’ and ‘how well maintainable’. Another connection is the effort it takes to create the automation. The resulting benefit is that you can execute your script over and over again.

The ideal situation: a test script with very small investment to create, used for a test that needs to be executed the whole life of the application and that doesn’t change overtime. No investment, no maintainability issues, maximum amount of executions. Result: maximum ROI.

Too bad, we’re not living in the ideal world. So we need to make some tradeoffs.

1. **No automation.**

   No need for maintainable test scripts, no automation investment. I have customers who use Microsoft Test Manager for test case management only, and they are happy with it. They maintain thousands of test cases and their execution, gathering information about the test coverage of the implemented functionality.

   In most situations, this is an ideal starting point for adopting Microsoft Test Manager and starting to look at test case automation. As a test organization, you will get used to the benefit of integrated ALM tools that support all kind of ALM scenarios.

2. **Shared Steps with Action Recording | Record Playback parts.**

   Collecting an action recording takes some effort. You have to think upfront what you want to do, and often you have to execute the test case several times to get a nice and clean action recording. So there is some investment to create an action recording that you can reuse over and over again. In Microsoft Test Manager you can’t maintain an action recording. When an application under test changes, or when the test cases change, you have to record every step again. A fragile solution for automation.

   Using Shared Steps (reusable test steps) with their own action recording solves this a bit.

   Find the test steps that appear in every test case, make a shared step of these steps and add an action recording to it. Optimize this action recording and reuse the shared step in every test case. This definitely improves the ROI. Now you can fast-forward all the boring steps and focus on the real test.

   The good thing is that when a shared steps changes, you only have to record that one again. Creating multiple shared steps with action recording and compose a test case is also a good scenario. After the zero investment, this is a good next step. You get used to the behavior of action recordings and have the benefit of reusing them throughout the project.
Action recordings of shared steps keep their value the whole project, there is some effort to create and maintain them but you will execute them for every test case, a good ROI.

3. Test Cases with Action Recordings | Full Test Case Record Playback.

The same activity as for the Shared Steps action recordings. But, you will use the action recording less and it is harder to maintain (more test steps). The ROI is thus much lower than in the Shared Steps situation.

The scenario where you create the action recording and execute often, for example on many different environments, will provide benefits. Microsoft Test Manager action recordings can be recorded on one environment and playback on another environments.

Another reason you might want to go with this scenario, is that you want to reuse the action recording for test script generation. See next step.

4. Generate test script from action recording.

This is a really nice scenario for quickly creating test automation scripts. See this How To video [6]. The maintainability of the generated code is hard. There are some tools in place to react on UI changes, which make it easier. With Visual Studio 2012 default the Code UI Map Test Editor is available to edit search criteria, rename controls and create methods.

Due to the UI Map XML, some common development practices like branching, merging and versioning of the test script are a challenging task.

In conclusion, the creation of test scripts generated from action recordings is really fast but hard to maintain. Together with the recording of the action recording (number 2), this influences the return on investment.

Conclusion

These five tips should make your testing effort more efficient. And they will not only work in agile projects, but they will work in all types of project. Only in agile projects you will feel the pain earlier when things go less efficient as planned. Not only these tips will bring benefits to your project, but I encourage you to create and try your own tips and improve them for every sprint.

Clemens Reijnen is a management consultant at Sogeti, specializes in Application Lifecycle Management, he gives ALM courses throughout Europe and is a frequent speaker at conferences. Next to supporting teams to deliver better software Clemens also writes on the blog www.ClemensReijnen.nl, where many articles and side projects can be found. His experience encompasses a deep knowledge and experience in software development.
Integration Testing from the Trenches

The concepts of integration testing presented with a casual formalism that's intuitive and fun.

By Victor Grazi

On the subject of testing, the variety of definitions you hear in the industry is as diverse as the blind men's descriptions of an elephant.

But in his important and thorough treatise “Integration Testing from the Trenches” Nicolas Frankel starts from basic definitions and develops the concepts of integration testing with a casual formalism that's intuitive and fun.

The fundamental premise is that Unit Testing is the testing of one thing, usually a method, in contrast to Integration Testing, which is the testing of two or more things. Therefore in unit testing, calls out by the method under test are preferably stubbed-in or mocked so that only that method’s algorithm is tested, in isolation.

This simplifies conceptually the notion of unit testing, particularly of how to test methods calling methods calling methods, because according to our definition, you don't call them at all, for then they would be “integration tests.”

The book posits early on that Integration tests are brittle and hard to diagnose, so they should not be used instead of unit tests, and that is the reason that continuous integration servers generally measure coverage of unit tests but not integration tests.

Most of the time, Integration Testing is either crudely defined, wrongly understood or imperfectly used. In order to maximize the ROI, the most important guideline should be: the larger the SUT (System Under Test), the lesser the required code coverage.

The book summarizes the first chapter with an honorable mention to a variety of related test categories:

- Mutation testing
- Functional testing
- Security testing
- Performance testing

Chapter two provides an excellent explanation of why integration tests can validly be ordered and how.

Unit testing frowns on tests ordering, for test cases should not rely on each other or on state. However, what goes for unit testing should not always blindly be applied to integration testing. In particular, having multiple ordered fine-grained methods with each method asserting the state of the system may be required for white-box testing.

The chapter covers JUnit in depth (using annotations introduced in JUnit 4.x), including the JUnit lifecycle callbacks and assertions, with detailed sample code.

It starts with a definition and brief discussion of the single assert rule:

Each test method should only provide a single assert, being the raison d’etre of the test method, or a tightly bound group of asserts, for example:

```java
Assert.assertNotNull(list);
Assert.assertFalse(list.isEmpty());
Assert.assertEquals(1, list.size());
```

This way, each assert is a little more restrictive than the previous and if the test fails, we will know exactly the reason (the list was null, it was empty or it had more than a single element).

The chapter also provides a nice treatment of various alternative frameworks to JUnit, discussing advantages over JUnit. The book covers some esoteric JUnit options for sequencings tests in a deterministic fashion, that many developers probably have not seen before:

JUnit provides some basic feature regarding ordering. One can choose to order by annotating the test class with org.junit.@FixMethodOrder and use one of the three following value:
1. org.junit.runners.JVM for no ordering at all
2. org.junit.runners.DEFAULT for deterministic but not predictable ordering
3. org.junit.runners.NAME_ASCENDING for ordering based on method name

Chapter 3 begins with a discussion of “Test-Friendly Design”, covering popular design patterns that make code more testable. Following that is a discussion of stubs, fakes, mocks, and dummies, and a code example that clarifies these in the context of dependency injection.

The books then segues into a list of various mocking frameworks including Spring Test and MockRunner for creating fakes, and then does a deep dive into the popular Mockito framework for stubs, mocks, and dummies. There is also a brief discussion of using Mockito for Behavior Driven Development, simulating “Given-When-Then” behavior.

In general, the book does not just list APIs, rather it explains them and then summarizes them with a list in a format that produces clarity.

In the next section, there is a discussion of fakes using the Spring Test framework, diving into Servlets and JNDI with ample code snippets. Spring Test conspicuously omits JMS, so the book next discusses MockRunner, which provides fakes for JMS, Struts, Servlets, Taglibs, JDBC, EJB, JMS, and JCA.

For each framework, the book covers options and discusses the decision tree for selecting the right framework for your use case.

The next chapter deals with a frank discussion on how to test real resources such as time for example mocking System.currentTimeMillis() and using Joda Time (now replaced with Java 8 Date Time API’s), files, databases (traditional and NoSql, e.g. MongoDB), mail, and FTP, all using fakes. A great treatment of the popular DBUnit framework is included. As is prevalent throughout the book, the chapter recommends best practices and best tools.

The chapter concludes with a discussion of when to use stubs and when to use fakes:

Having more than one option always bring the quandary of choice. In most contexts, it is better to use the Fake server as it
1. requires less effort and
2. makes code more readable.

It is recommended to use the Stub server only if testing requisites really require it.

Subsequent chapters deal with a variety of topics including Web Service Integration (SOAP and REST) and associated tools and faking frameworks, Automated Testing (Build tools such as Maven and Gradle, Continuous Integration servers such as Bamboo, Hudson, and Jenkins).

The book concludes with a reference appendix for all of the frameworks and tools mentioned throughout the book.

I found after reading this book that some of the fear of the unknown that I normally experience when embarking on test creation has evaporated.

This is an important and thorough book that covers all facets of integration testing, with a large overlap of unit testing and other testing techniques.

It is a must have reference for all software developers and project managers using any language. It is clear, easy to read, and comprehensive. The book is replete not just with APIs but with best practice advice as well, as well as many helpful examples.

(This article was originally published on http://www.infoq.com/articles/Book-review-Integration-Testing-from-the-Trenches)

Victor Grazi is the Java queue lead at InfoQ. Inducted as an Oracle Java Champion in 2012, Victor works at Nomura Securities on core tools development, and as a technical consultant and Java evangelist. He is also a frequent presenter at technical conferences. Victor hosts the "Java Concurrent Animated" open source project on SourceForge.