



MANCHESTER , UK  
21 - 24 November 2011

# Test Strategies in Agile Projects



**Anders Claesson**  
[anders.claesson@enea.com](mailto:anders.claesson@enea.com)

Agile development methods are becoming more and more common in our projects. The objective is to achieve higher quality and shorter lead times with minimal overhead. A well defined agile test strategy guides you through the common obstacles with a clear view of how to evaluate the system. Testing starts with the exploration of the requirements and what the customer really wants. By elaborating on the User Stories from different perspectives, both efficient and effective test ideas can be created.

## Contents

1. Introduction	2
2. Experiences	3
3. Workflow	3
3.1 Project Initiation	4
3.2 Release Planning	10
3.3 Sprint Iterations	20
3.4 End Game	39
3.5 Release	42
4. Biography	45
5. References	46

## Abstract

Agile development methods are becoming more and more common in our projects. The objective is to achieve higher quality and shorter lead times with a minimum of overhead. With frequent deliveries, close cooperation within the agile teams and the customer, continuous integration, short feedback loops and frequent changes of the design creates new challenges but also new opportunities for testing.

A well defined agile test strategy guides you through the common obstacles with a clear view of how to evaluate the system. Testing starts with the exploration of the requirements and what the customer really wants. By elaborating on the User Stories from different perspectives, both efficient and effective test ideas can be created. Testing becomes a continuous and integrated process where all parties in the project are involved.

# 1. Introduction

Since the popularity of Agile development methods have increased over the years, it is more important than ever to adapt how we do testing in the most efficient and effective way.

In Agile projects the software is developed in an iterative way where requirements and solutions evolve through collaboration between self-organizing cross-functional teams.

Rapid delivery of high-quality software requires discipline, teamwork, skill, communication and continuous quality assurance throughout the whole process.

A key principle of Agile development is the recognition that during a project, the customer is likely to change their mind frequently about what they want and what they need.

With short iterations, feedback and adjustments can be made, not only to the product, but also to the way we do our work via retrospectives. During the retrospective meeting we can look at how the work has been done so far, learn from mistakes and suggest how to make it more efficient.

Software testing is defined as an empirical investigation conducted to provide stakeholders with information about the quality of the product or service under test, with respect to the context in which it is intended to be used.

Since the prerequisites change quite frequently in an Agile environment a different approach is needed compared to the old school of software testing (with a more sequential way of working). The main goals of software testing are however much the same. That is to find bugs, to show that

the system works, that it is fault tolerant to a certain degree, to provide information about the risks of using it, from a user and an operational perspective, and to prevent faults.

With an effective Agile testing strategy the team and the whole project will be guided through each step in the development process with a clear view on how to achieve good enough quality. A holistic approach is used where everyone is involved in one way or another (not only the testers) to assure sufficient quality is reached in each step of the process.

The testers and the designers are guided through the strategy on how to evaluate the product in the most efficient way by the suggested methods, techniques and tools that form the basis of their work. It is not a prescriptive way of working, but more of a constructive and creative way of how to think, learn and evaluate what is (from many different perspectives). In the end we all want the customer back, not the product. So we better find those annoying bugs before the customer does.

TWEETABLE



**A well defined agile test strategy guides you through the common obstacles with a clear view of how to evaluate the system.**

TWEETABLE



**A test idea represents a question you want to ask to explore a component or system under test, in order to find out how it really works, and if its behavior may be considered correct/ acceptable or not.**

## 2. Experiences

Since working with testing for nearly three decades I have seen many development models rise and fall in popularity. I have worked with datacom systems, telecom, avionics, train signaling, Client/Server, and small and large applications and systems.

In the beginning the V-model was put on a big chart outside the restaurant for the employees in the company I was working for then (1982). A couple of years later RUP (Rational Unified Process) became the fashion of the month. Both of these early models were very time consuming to use from a testing point of view since they required so much documentation.

In reality, my colleagues and I were adopting work methods which have now become widespread and known as Agile software development. We were already doing exploratory testing, since the documentation we had as a basis for our tests did not match reality anyway (only to some extent). My worst experience was when I wrote a test specification 1,000 pages long (it was also fully automated using an in-house scripting language). It took a lot of effort to write and, of course, did not work in the lab.

Experiencing writing test charters instead is such a relief from what I did in the old days. Now it is possible to actually do some productive testing work. The methods described in this document have evolved over a long period of time, also inspired by a number of people leading the evolution of testing practices in the direction of becoming more of a science than an art.

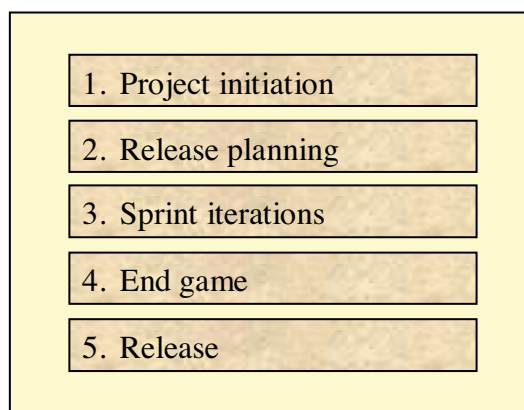
By using the cumulative testing knowledge from the whole testing community without too much bias towards only one school of thought, but with an open mind, a lot of new insights can spread over several directions.

How testing can be done in the most efficient way (by getting the best parts from each area) and to continue the journey, expand the concept of how to ensure that we build the best software systems possible for the coming future.

## 3. Workflow

The test strategy should be created in an evolutionary way during the whole project. The picture below shows the main steps of an Agile development process [Crispin / Gregory - 09].

Scrum, which is a common iterative and incremental process of software development, is used as a basis to represent an Agile way of working.



On page 4 there is a short overview of the different steps in the Agile development process where the testing strategies are included:

### 1 - Project Initiation

Find out what the goals with the system are, who the customer and users are and what problems the customer and user want to solve with the system. Identify all the requirements for the system.

## 2 - Release Planning

Decide contents and dates for all releases and demos for the customer.

## 3 - Sprint Iterations

Identify all design and testing tasks. Estimate what and how much testing is required. Perform structural, functional and characteristics testing. Make a demo for the customer at the end of the Sprint. Have a retrospective meeting afterwards to evaluate how the work was carried out. Learn from both good and bad experiences. Use what you learned in the next iteration.

## 4 - End Game

Perform a complete regression test of all delivered functionality. No new features are allowed at this stage. Test the most important characteristics in a complete environment, such as (final testing of) performance, reliability, stability, security, scalability, etc. Perform a user acceptance test.

## 5 - Release

Check against the release criteria that everything has been done and is ready for release. Release the system to the customer and hand it over to your support and maintenance organization. Finish with a project retrospective.



TWEETABLE

**The most common (and difficult) problem when specifying the requirements is missing information. The purpose of an Agile way of working is to gradually fill in the missing parts while the development progress is in each Sprint.**

# 3.1 Project Initiation

In Scrum there are three main roles to be defined at the start of the project.

- > **The Product Owner** identifies and prioritizes the features to be included in the Sprints.
- > **The Scrum Master** is responsible to ensure that the Sprint stays on course. Also to facilitate the Sprint planning and that everyone follows the Scrum rules and practices.
- > **The Team** is responsible for developing (to analyze, design, code, integrate, test and document) the required functionality and characteristics. Teams are self-managing, self-organizing and cross-functional.

To be able to start on any test strategy work you need to understand what the project is all about. Therefore you need to ask some relevant questions. The following list is based on [Cagan -06].

- 1) Why are we doing this project?
- 2) Exactly what problem will this solve? (value proposition)
- 3) For whom do we solve that problem? (target market)
- 4) How will we measure success? (business metrics)
- 5) What alternatives are out there? (competitive landscape)
- 6) Why are we best suited to pursue this? (our differentiator)
- 7) Why now? (market window)
- 8) How will we deploy this? (deployment strategy)
- 9) What is the preliminary estimated cost? (small/medium/large)
- 10) What factors are critical to success? (solution requirements)

These are typical questions the product

owner should answer before launching the project.

This is also important to understand from a testing point of view, both as a motivator, and to get the bigger picture of the project. The success factors should be guiding the test strategy to assure they are verified and accomplished regarding the product and its quality.

The product owner produces a so called product back log at the start of the project, which includes the high level requirements for the project. The product backlog is the main input for the initial (pre-) planning conducted by the team.

#### Pre-planning tasks:

- Make an overall estimation and prioritization of the included backlog items.
- Identify impediments, and plan for how to remove them.
- Identify lessons learned from previous releases or iterations that the team wants to incorporate in this release.
- Review and revise the team's standards and practices.
- Make an initial technical risk analysis.

To gain a better understanding of what is to be developed it is useful to view the system from three different perspectives [Claesson3 -04]:

#### User: What does the user want to do with the system?

Analyze the requirements, features, needs, expectations, usage profiles, user data, operational scenarios, User Stories, interoperability, usability, safety, compatibility and customer values.

#### System: What should the system be capable of doing?

Analyze the functions, system characteristics, interfaces/interactions, operational environment, availability,

robustness, scalability, maintainability, etc.

#### Risks: What problems may occur?

Consider potential risk areas and where you and the developers think there might be problems. Analyze potential problems regarding the operation of the system, failure mode handling difficulties, instability, inconsistencies, incorrect functional behaviour, insufficient characteristics, weak specifications, complex areas and faulty areas.

Elaborating and discussing the three perspectives described above will give valuable input on how to start forming the test strategy. At an early stage it might not be possible to answer all of these questions, but they can be used to guide the following discussions in the team about the direction of where, what and how to test in different areas.

It is also very important to find out as early as possible what the customer values the most regarding the quality attributes of the system. Typical quality attributes are performance, availability, reliability, etc. [ISO/IEC 9126-1], [Testeye -10], [Al-Qutaish -10], [Ganesh -10], [Wikipedia1 -11] (there are more than 50 quality characteristics of a typical software system). These are often forgotten or neglected in rapid development projects, but nevertheless very important for the customer, the user and the supplier.

For each quality attribute:

- Identify how important it is for the user/customer by a relative weight in percent (100 % is divided among the quality attributes the customer identifies as important).
- Identify how satisfied the user/customer is with the current version of your system in operation (if there already exists any previous versions of the software in service).

A relative value is given (by asking a selected



User/Customer Value Analysis					
Quality Attribute	Importance Weight %	Customer Satisfaction	Priority IW / CS	Risk/Cost of failure	
1. Reliability	30	0,9	33	High	Critical
2. Availability	15	0,8	18	High	Critical
3. Usability	20	1,1	18	Medium	High
4. Performance	20	1,1	18	High	Critical
5. Service Level	10	1,0	10	High	High
6. Maintainability	5	0,8	6	Medium	Low
Summary:	100 %	X=0,96			

Supplier costs

Customer costs

X < 1: Less satisfied  
X > 1: More satisfied

number of your customers) where 1 means “OK”, < 1 means “less satisfied” and a value > 1 means “more satisfied”.

By dividing the importance with the current satisfaction level, a priority weight can be calculated. The prioritized quality attributes can also be combined with a risk assessment (what is the probability of this specific quality aspect failing in operation and what are the consequences, e.g. if the system response time would be 10 minutes instead of less than a second).

By knowing more about what the customer and users value about your product will help a lot, both in design and testing. Consider which attributes are most important from a customers’ point of view and a suppliers’ point of view (e.g. maintainability, portability). Not so important attributes may be tested less but still covered to some extent.

As soon as the product backlog is defined the refinement of each high level requirement can begin. In Scrum so called “User Stories” are used to define the requirements. A “User Story” is a software system requirement formulated as one or two sentences in the everyday or business language of the user. The purpose of writing User Stories is to make the requirements easier to understand and discuss.

Examples of some User Stories:

- 1) As a **[registered user]**, I want to **[log in]**, so that I can **[access subscriber content]**.  
(A functional requirement, this is the most common type of User Story)
- 2) As a **[normal user]**, I should get **[new screens to appear within one second or less in 90% of the transactions, where the maximum response time for a new screen to appear is less than 8 seconds]**, so I can **[use all features without any longer delays that would otherwise disturb me in the use of the application]**.  
(A performance requirement. These kind of requirements usually “disappear” or are not included at all because they are more difficult to specify – Not a good idea! Performance needs to be carefully specified because it affects the whole system architecture)
- 3) As a **[normal user]**, I do not want **[any single point of failure in the application to cause any loss of data]**, so that I would need **[to retype it again]**.  
(A reliability requirement. This is actually a negative User Story of things I do not want to happen)

- 4) As a **[customer]**, I want to **[quickly find and purchase an item with a minimum number of steps]** so that I can **[continue shopping for the next item or leave the website]**.

(A usability requirement. Unverifiable phrases may need to be elaborated and if possible quantified - what would “quickly” mean for a typical user customer?)

- 5) As a **[normal user]**, I want to **[be able to write a letter]**, so that I can **[read and edit it on all versions of Windows from Windows XP and Office 2003 to Windows 7 and Office 2010]**.

(A compatibility requirement. This could be very important for a normal user if he/she works as a consultant for example, using different computers with different versions and platforms for word processing when writing on the same document).

The User Stories are the main input for all testing. The tester’s role is to question what is written in order to clarify it. Also to assure the acceptance criteria is complete enough.

The user perspective can be evaluated on a continuous basis since you involve the user/customer during the design. Generally, software requirements are a communication problem.

User Stories may be written on cards with the story and conversations for clarification

on one side and the confirmation on the back of the card [Jeffries -01], [Marekj -08], [Kelly -08].

**Card:** Short description of the story, “as a user, I want...”

**Conversation:** Notes and/or an illustration with explanations

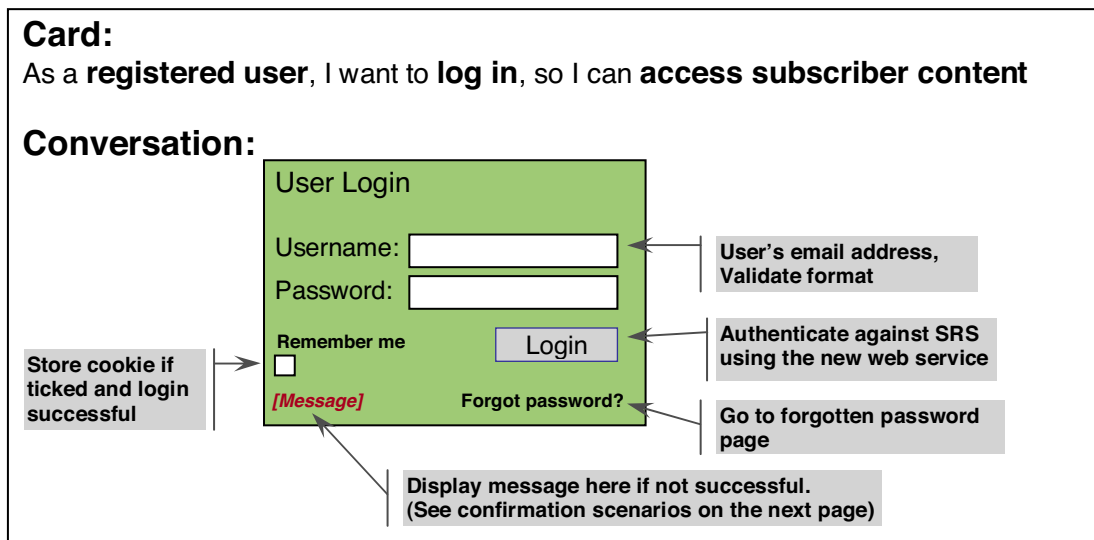
**Confirmation:** Acceptance criteria. The tests that must pass in order to validate that the User Story has been delivered and works as intended.

Stories for characteristics need to be quantified (as complete as possible) to be testable. Example:

“As a user I should get a response from the system on any request within one second or less in 95% of all cases (up to 85% of the system capacity regarding processor utilization and 90% RAM memory usage), where no response takes longer than 8 seconds.” If the system load exceeds the maximum limit regarding resource usage and response times an error message should be sent for any new requests which are also noted in the system log (can be included in the acceptance criteria).

More details may also be expressed as additional User Stories.

Example of a more complete specification of a User Story. The example is based on (Kelly -08):





### Confirmation:

- Success** Valid user logged in and referred to the home page
- a) Valid user name and password
  - b) "Remember me" ticked – Store cookie/automatic login next time
  - c) "Remember me" not ticked – Manual login next time
  - d) Password forgotten and a correct one is sent via email

- Failure** Display message:
- a) "Email address in wrong format"
  - b) "Unrecognized user name, please try again"
  - c) "Incorrect password, please try again"
  - d) "Service unavailable, please try again"
  - e) Account has expired – refer to account renewal sales page

Both the conversation and the confirmation are part of the requirement, i.e. the acceptance criteria also becomes part of the specification of how it should work. Note that "b" and "c" under "Failure" are invalid error messages since the system should never respond to which one of the password or user name was incorrect separately for security reasons (this is a defect which becomes visible here when the User Story is specified, if detected before the designer starts with the implementation).

When analyzing the User Stories, make sure all types of requirements are covered. The following lists is based on [Wiegers -03], [Wiegers -10].

### Functional requirements

The externally observable behavior (black box view of the system boundaries) with user actions as input operating under certain configurations, conditions and data.

### Characteristics requirements

Indicates how well the system should perform specified behavior. The system characteristics may also be referred to as quality attributes. Common characteristics are performance, reliability, usability, etc. The standard [ISO/IEC 9126-1] can be used as a source for which characteristics to use.

### User Stories and scenarios

An interaction between an external actor/ user and the system, in order to achieve a specific goal or business task. A scenario

represents a single path through a sequence of related User Stories, using a defined set of input data.

### Business rules

Certain user classes that can only perform an activity under certain conditions. Functional requirements may be derived from the business rules to enforce the rules. Business rules might be that a certain calculation method should be used.

### External interfaces

Describe the connections between your system and the outside world. Divide the external interfaces into different types (user interface, interface to other systems, etc.).

### Constraints

Any design and implementation constraint that will restrict the options for the developer. Devices with embedded software must often respect physical constraints such as size, weight and interface connections.

### Data definitions

The customer may describe the format, data type, allowed values, default values for a data item in the composition of a complex business data structure (e.g. the Zip code in the mail address is only five digits).

The most common (and difficult) problem when specifying the requirements is missing information. The purpose of an Agile way of working is to gradually fill in the missing parts while the development progress is in each

Sprint. Both discussions within the team and the customer representative may provide clarity when in doubt. But you need to know what to ask for. Below are some examples of information that is usually needed, but not defined:

- Missing boundaries of input or output values.
- Characteristics which are difficult to quantify or even specify.
- Missing actors and interfaces.
- Undefined exceptions and error handling.
- Undefined system limits (e.g. performance limitations, configuration limitations, number of users, etc.).

The next step in the process is to make a test analysis to assure all User Stories are testable and complete. Then it is possible to evaluate the function, characteristic or system by the use of any known and available test method, test technique and/or test tool, at a reasonable cost and effort to achieve a good enough quality assessment.

Consider the following when analyzing the requirements:

- 1) Who are the customer(s) and the target user group?
- 2) Why does the orderer/customer want the system?
- 3) Which user/usage goals should be met?
- 4) What user problems should be solved?
- 5) Which user benefits should be achieved?
- 6) Which functions and characteristics are included?
- 7) What is supposed to happen (what is the story)? What sequence of actions? Is it connected to an overall (business) process?
- 8) When should it happen?
- 9) Where should it happen?
- 10) Why should it happen? What is the purpose, goal and underlying need?
- 11) How will it happen? In what context? What characteristics apply and to what extent?

- 12) What are the most common and critical parts of the functionality from the users point of view?
- 13) Are there any performance requirements included?
- 14) What is an acceptable response time for the users?
- 15) How tolerant should the system be to faulty input or user actions?
- 16) What are the limitations in the software and/or hardware regarding characteristics, features, functions, data, time and space? Which exceptions are assumed directly and/or indirectly?
- 17) Is the description complete enough to proceed and decide how to design, implement and test the requirement, involved features and the system as a whole?
- 18) Which problems and risks may be associated with these requirements?

Based on the given answers from the above questions and previous analyzes, it is now possible to start writing down test ideas and also start to create a framework, as a basis, for the overall test planning and strategy.



— TWEETABLE —

**To gain a better understanding of what needs to be developed it is useful to view the system from three different perspectives.**



— TWEETABLE —

**The User Stories are the main input for all testing. The tester's role is to question what is written in order to clarify it.**

## 3.2 Release Planning

The product owner and the Scrum Master make the overall planning in the project.

The test plan should be made on an overall level, to cover all test activities from inspections, test modeling, unit testing, integration testing, functional testing, characteristics testing to acceptance testing. Also all support activities such as building the test environment, version control of the test environment, incident reporting, etc. should be included.

The Agile testing strategy is part of the overall test planning.

The Test Strategy can be seen as a long-term plan of action. If your organization wants documentation about your overall test approach in projects, consider taking this information and putting it in a document that doesn't change much over time. There is usually a lot of information that is not project specific that can be extracted into a general Test Approach document.

This document can then be used as a reference and needs to be updated only if the process changes. It may include how to build a test environment, how to do incident reporting, etc.

The test plan should be written on a high level to begin with, to identify the main testing activities and the needed infrastructure for testing. Details are added in each Sprint based on the contents of the Sprint.

- > Testing is included in the project planning (Release & Sprint-plan).
- > The test plan and strategies evolves within each iteration.

- > The High Level Test planning is made at "Sprint 0".
- > Sprint test planning is made by the team.
- > A separate test Sprint may be needed in the end for non-functional testing, to verify corrections and for final regression testing.
- > If you have a large and complex system you may also need to have a separate team for system testing. The main part to be tested in the system test is characteristics, e.g. performance, which requires a lot of planning, preparation and tool support. You may also need expertise either within your team or hired from outside (e.g. security experts) for each characteristic to be tested.

During the release planning stage you should start to define the overall testing goals, strategies and follow up procedures to guide the project and the team in the right direction towards quality.

### 1) Define the test scope.

Identify what is included and what is excluded in testing for this particular project. Consider what is new, what has been changed or corrected for this product release?

### 2) Identify which levels and types of testing are needed.

Look at what system integration levels are needed (e.g. module, function, sub-system, system), test types (e.g. structural, functional, characteristics) and test objectives (e.g. acceptance test).

### 3) Define test goals.

Define the testing goals based on the project goals and the analysis of the system to be built. Testing goals are used to give directions of what testing needs to achieve in the project.

#### 4) Define test strategies.

Identify efficient test strategies for different areas (e.g. how to find out what to test, the use of test automation) to reach the test goals. The testing strategies should be used to give directions for all types of testing. The strategies may suggest which test techniques and tools might be useful, how to select test data, how to build and configure the test environment and how to prepare and run the tests.

#### 5) Specify test metrics.

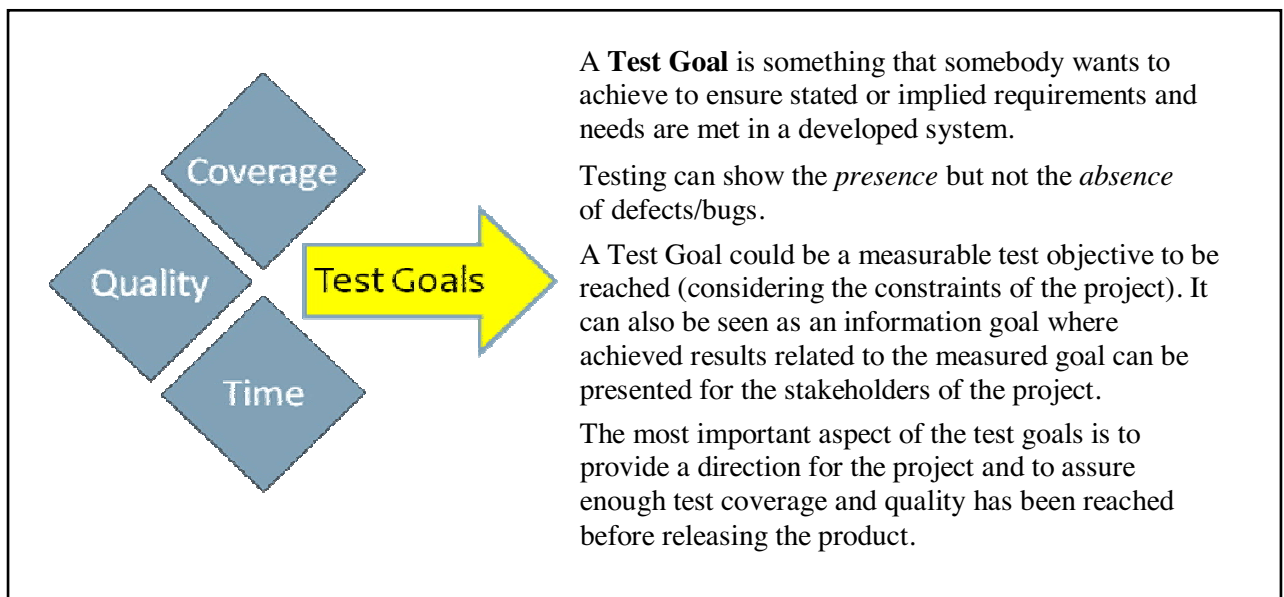
Define test metrics to be used for follow up of the testing and the associated goals. On the task board you may only see what has been done but not how much test effort has been spent in different areas, the actual test coverage and how many faults were found.

#### 6) Specify the “Definition of Done”- parts related to test and quality.

The “Definition of Done”(DoD) is specified and used as a checklist by the team to determine when activities/tasks performed during the system development can be considered finished.

#### 7) Define when to stop testing.

Define when to continue or stop testing before delivering the system to the customer. Specify which evaluation criteria is to be used (e.g. time, coverage, quality) and how it will be used. In Scrum a so called acceptance criteria is often used, that is defined by the customer. The acceptance criteria are used in the final acceptance test to decide whether the system is ready for delivery, installation and production/ service.



#### Coverage goal examples:

1. To cover all requirements (User Stories) with at least two Test Cases (one positive and one negative) or more depending on the size of the problem to be solved, its complexity and the associated risks.
2. To cover how functions and features are implemented and how they are used (from an internal and external

system perspective). Interfaces, protocols, connections to third party products, external hardware, external sensors/devices for measurements and connections to other systems/sub-systems. Functions which are vital for the system such as start/restart is especially important to cover.

3. To cover all characteristics considered most important for the system such as

performance, stability, load, security, etc.

- 4. To cover all system parts, i.e. sub-systems or other systems, or interfaces, both user and system interfaces.
- 5. To cover each code unit to a level decided by the risk analysis and any safety regulations, i.e. 100% statement coverage, 100% branch coverage.
- 6. To cover all documentation either by inspection or by using the documentation, i.e. to use all parts of the user’s manual.
- 7. To cover all high priority risks.

- means the designers are doing something fundamentally wrong in their design process.
- 3. At the release date there shall be no unsolved high priority risks or critical Incident Reports open. Planned test coverage should have been achieved.
  - 4. Characteristics measurements should be within defined limits in, for example, response times for a defined set of users and configurations.
  - 5. The pass rate for the acceptance tests should ideally be 100%, but exceptions may be accepted depending on the severity of the found faults.

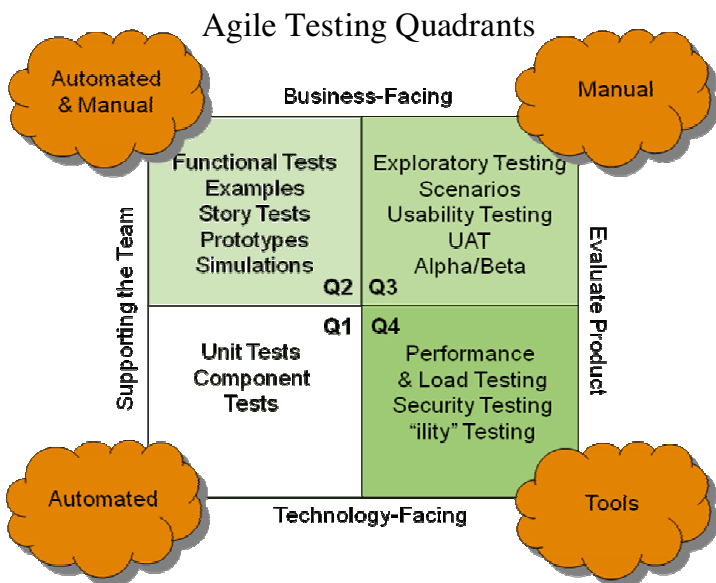
Quality goal examples:

- 1. To reach an acceptable limit of remaining faults and severity level, for example: No blocking faults, one major and only a few minor ones as long as the functionality, stability, security and performance is not affected.
- 2. The failure rate should be decreasing when executing new tests. The objective is to reach a certain level of number of defects found per week/hour of testing. If the failure rate is constantly high it

Time goal examples:

- 1. The planned completion date has been reached.
- 2. The burn down chart deviations compared to the planned curve is within acceptable limits, for example 10%.
- 3. Incident report turnaround times for correction handling in each category of A, B and C are within the number of days stated by the team.
- 4. Planning precision is within acceptable

Test Strategies





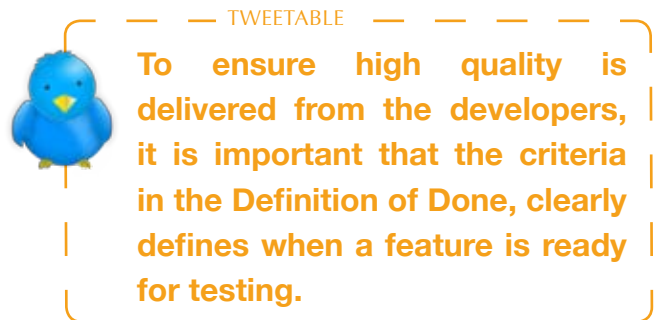
limits regarding time, scope/contents of the Product Backlog, Sprint Backlog and the set of features delivered compared to initial plans. Also consider whether the velocity of the team is sustainable and predictable.

Agile testing can be divided into four different quadrants to show different aspects of testing and purpose, [Gregory -10]. The Agile Testing Quadrants was originally created by Brian Marick [<http://www.exampler.com/>]. The following description is based on [Crispin - 09].

- Q1 Represents a Test Driven Development practice. Unit tests verify the functionality of a small subset of the system, such as an object or method. Component tests verify a larger part of the system, such as a group of classes that provide some service. Both types of tests are usually automated.
- Q2 Represents tests that support the work of the development team at a higher level. These business-facing tests, also called customer-facing tests define external quality and features the customer wants. Tests are oriented toward illustrating and confirming desired system behavior at a higher level.
- Q3 Represents business-facing examples to help the team to design the desired product. When the programmers write code that makes the business-facing tests pass, they might not deliver what the customer really wants. The focus is to test how a real user would use the product.
- Q4 Represents tests focusing on the characteristics of the system such as performance, robustness and security. Tools are often required, which can be used to create test data and for load and performance tests of the system.

Some **basic principles** for Agile testing [Claesson4 -11]:

- > Testing is performed early and often.
- > Testers are included in a cross functional development team.
- > “User Stories” are tested.
- > Close cooperation with developers and customers.
- > Continuous integration and regression tests.
- > All test results are logged.
- > Defects are reported.



## Useful methods and techniques:

### Requirements Based Testing

Cover all requirements with Tests. To avoid the risk of testing only the normal/positive cases you may benefit from using tools like [[http://www.glocksoft.com/resource\\_viewer.htm](http://www.glocksoft.com/resource_viewer.htm)] to extract all fault codes that the programmer actually has inserted into the code (but “forgot” to mention in the user manual).

### Design Based Testing

Cover how functions and characteristics are implemented (from an internal system perspective). Interfaces, protocols, connections to third party products, external hardware, external sensors/devices for



measurements and connections to other systems/sub-systems.

### Risk Based Testing

Identify potential problems early (by a technical Risk Analysis) and focus testing in these areas.

### Exploratory Testing

Simultaneous learning, test design and test execution. Testing where the tester actively controls the design of the tests as those tests are performed and uses information gained while testing to design new and better tests, [Bach2 -01].

### Error Guessing

Anticipation of defects in certain areas based on experience.

### Taxonomy Based Testing

Classification of defect types. Used as a basis of what common problems to look for when testing.

### Attack Based Testing

Attempting to force specific failures to occur.

### Model Based Testing

Create models of the system behavior, e.g. state diagrams or scenarios from an external actors' point of view (User Stories). The models can be used to generate test cases.

### Scenario Based Testing

To test a flow of consecutive actions directed by a specific usage goal or failure modes. A scenario is a hypothetical story, used to help a person think through a complex problem or system. It is usually based on User Stories, use cases or operational scenarios.

### Combinatorial Testing

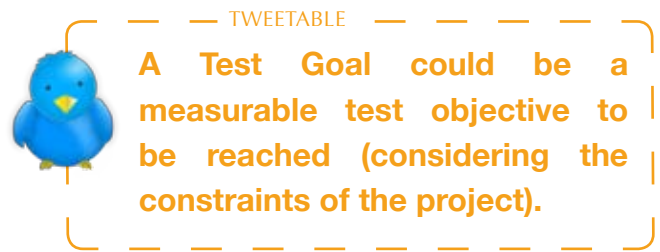
All possible discrete combinations of each pair of input parameters to a system are tested (also called pair-wise or allpairs testing. Can also be "n-wise" including 3-wise, 4-wise, etc.), [Grindal -07].

### Value Based Testing

To cover the most important customer and user values whether expressed or not (i.e. functions/features and/or characteristics valuable for the customer and user). A user value might be that "the response time of the system is less than one second when navigating through the menus" (which could be based on a performance requirement).

### Prototyping

To participate in the human interaction design by asking questions and making models (which can be used when doing model based testing).



### Definition of Done

Definition of Done is a simple list of activities (writing code, coding comments, unit testing, integration testing, release notes, design documents, etc.) that add verifiable/ demonstrable value to the product. Focusing on value-added steps allows the team to focus on what must be completed in order to build software while eliminating wasteful activities that only complicate software development efforts.

To ensure high quality is delivered from the developers, it is important that the criteria in the Definition of Done clearly defines when a feature is ready for test. The most important parameters in Agile Development are quality and time. That is to deliver high quality rather than many features within a limited period.

Example of different criteria to determine if a test activity can be considered "Done" or not:

### Structure based testing

- 100% Software Design and Module Specifications covered.
- 100% Statement Coverage of the source code. If 100% is not possible to reach, a desk check can be performed for the remaining parts. If you are testing a safety critical system you may need higher levels of code coverage, such as branch coverage (level B) or Modified Condition/Decision Coverage (for level A software according to DO178b safety standard for Avionics software).
- Boundary Value Analysis Testing performed.
- Equivalence Classes and Input Partitioning Testing performed.
- All Tests passed and no remaining faults to be corrected.
- All code reviewed.
- Known weaknesses described.
- Component testing reported (including obtained test coverage).

### Integration testing

- Internal and external interfaces in the sub-system covered by verifying protocols and syntax in all messages.
- More than 40% of all tests are negative test cases. The main part of the source code, at least for real time applications, is intended for error handling of some kind. Therefore a lot of negative testing is needed to obtain sufficient coverage also on an integration and functional level.

### Functional testing

- 100% requirements coverage.
- 100% coverage of the main flows in the operational scenarios.
- 100% of the highest risks covered.
- 100% of externally observable system states covered.
- 100% of externally observable failure modes covered.
- Operational manuals tested.
- All failures found are reported.
- Boundary Values, Equivalence Classes

and Input partitioning testing made for all input data.

- All combinations of input and output parameters and values covered (pair-wise coverage).

### System testing

- End-to-End tests covered including all features and functions.
- Scenarios with the most common, most used and most critical paths covered. Also less frequent and common failure cases covered to some extent.
- The most important characteristics of the system covered.
- Testing is done in a complete (customer/ production like) environment with all HW and SW included (as much as possible).
- All system interfaces (also with other systems) covered.
- All high priority risks relating to system level issues covered.

### When to stop testing

To decide when to stop testing and deliver the product to the customer could be a difficult task. There is always another test that could be run to assure the system is good enough. To define the test stop criteria in advance will help the team to focus on what needs to be achieved in the end before the system can be released. The Definition of Done is a great help during the Sprint iterations, but the test stop criteria is the final goal to be reached regarding the quality of the system. The test goals set the direction and should also be directly connected to the test stop criteria. Some examples:

### Coverage

All parts of the requirements, needs, expectations, user/customer values, functional behavior, code, data, scenarios, characteristics, functional-/ system-/ and operational configurations and combinations, user profiles, risks, failure modes, hazards and user/system documentation should be

covered. The extent of the required coverage is determined by what is new/changed and the associated risks of failure.

### Quality

Testing should stop when the probability of remaining faults is reduced to a level that can be accepted by the customer/user [Beizer -90].

To evaluate the quality the following should be considered:

- The fault intensity (i.e. how many faults are found per week).
- The fault density (the number of faults found compared to the number of functional requirements, functional parts, complexity, scenarios and characteristics are tested).
- The consequences of found faults (what is the severity level and system impact for the user/customer).
- The current risk level (is it within acceptable limits?).
- The error trend (what predictions can be made regarding how many faults that could remain in the system?).

### Time

If the agreed delivery date has been reached the business aspects also need to be considered. What is most important: time to market, feature set, defect levels or the expected return on investments for the market window of the system?

All parts of the Definition of Done should also be fulfilled. If the project has agreed to deliver a certain number of functions and features at a certain time, this will also affect the decision to continue or stop testing. If less functionality has been developed from what has been planned the project needs to negotiate with the customer, also considering the current level of quality and risk whether to release or not.

### Integration and Coordination

When planning the contents of each Sprint, the individual dependencies between

functions, features and system parts (for example third party products) needs to be identified. The objective is to be able to demonstrate everything that has been developed within the Sprint at the Sprint demo. Ideally each feature should be as independent of other features as possible.

When integrating the system the order in which functions and features are developed is very important. To visualize the dependencies a system anatomy map can be used to plan and control the interrelationships between all functions and features in the project.

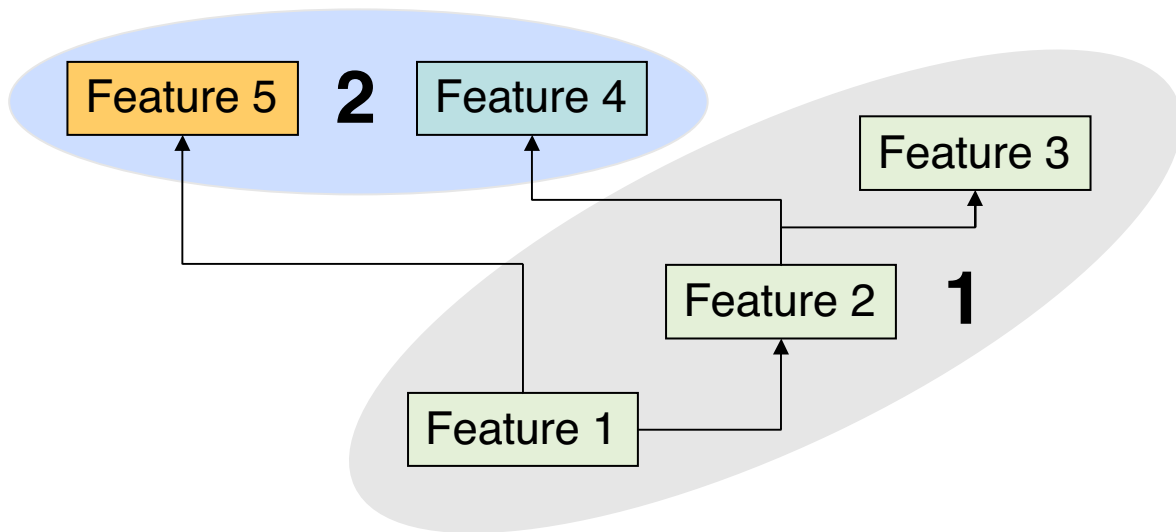
The anatomy shows all features/functions and their dependencies to other features and functions in a hierarchical and logical way. Ideally functions/features should be integrated up to a level where there is a test interface (or real interface) to access the feature. It is of little use if designers deliver features impossible to reach until the last increment when the project usually is in a very tight time schedule and there is little time left to do any testing.

Overleaf is an example of a simplified anatomy map showing the relation between the features to be developed in the first Sprint and the features to be developed in the second Sprint.

Basic functions such as start/restart should be delivered early.

The anatomy shows the total functionality of the system. Try to keep it simple or divide the chart into more than one page showing the main parts on the top page with details regarding dependencies on the following pages.

The anatomy map will immediately show the consequences of delayed or faulty features. Its dependencies give a good basis for how to re-plan and focus on what is possible to test and when.



The analyzed risk level per feature can also be shown in the anatomy map to make designers, integrators and testers prepared for possible upcoming problems. The higher risk on the feature the earlier it should be delivered to have enough time to evaluate and correct the problems.

Integration driven agile projects are usually possible to run at a faster pace than “normal” agile projects (if it is a large project involving many teams) where the teams (usually the designers) and the product owner decide the order of when User Stories/Features should be implemented and delivered.

The reason is that whole testable features (by using vertical integration) can be delivered much earlier which makes it possible for the testers to give faster feedback to design on found faults or problem areas. In integration terms it is called “Thread integration.” The disadvantage is that SW units may need to be opened several times before all changes in the project have been implemented (which the designers want to avoid because it takes them more time than just opening and working on a SW unit once).

It is also essential from a system test perspective to have clear synchronization and integration points where branches are merged from different sprint teams and put

together to allow the system testers to start evaluating the performance and end-to-end scenarios as soon as possible. Otherwise that will not happen until the last sprint or the last week before the final acceptance and delivery to the customer (which is far too late if serious performance issues show up that need to be corrected).

From the anatomy map an estimation of how much design/test time is needed per feature and when. Resources can be utilized much more efficiently and effectively.

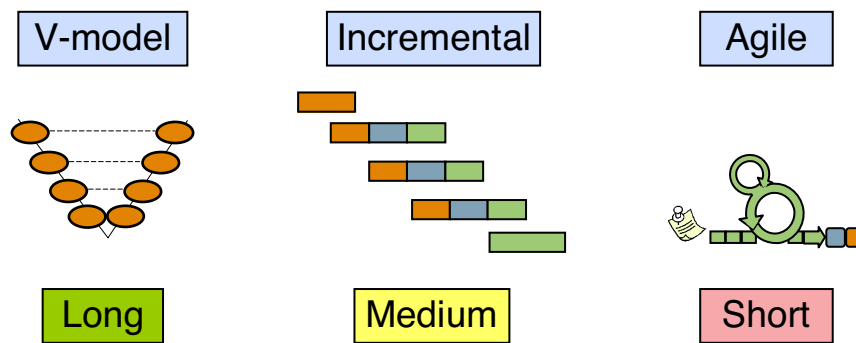
Stubs and/or drivers may need to be designed as well to be able to reach all parts for testing purposes.

Recommended integration strategies:

- > Try to make the deliveries in the sprints as independent as possible of other sprints and sprint teams.
- > Try to plan the contents in each sprint according to the anatomy, and as close to the calculated velocity and capacity of the teams as possible.
- > Plan the sprints based on the acceptance scenario from a customers’ point of view.
- > Make a risk analysis of the sprint plan.

- > Keep the system anatomy under configuration control.
- > When negotiating the contents of the anatomy use yellow sticky notes (one function or feature per note) on flip chart sheets (to be able to bring back for documentation) and move around the features until you find the most optimal solution.
- > Log issues and problems during the anatomy negotiation.

### Planning horizon when using different models for system development:



The need for test coordination increase with the size of the project and the level of short term planning. In Scrum and Agile development the idea is to plan only one or two sprints ahead, with some exceptions for system related parts and structures. Many projects have more than one Scrum team. In some cases Scrum teams are also located/outsourced to other countries.

In agile projects the teams are cross functional. The following three main problems may occur with cross functional teams [Hazrati -10]:

- > Sub-optimization at the project level.
- > Inefficiencies due to lack of coordination across the project.
- > Reduced expertise because of limited knowledge sharing across specialists.

One example of sub-optimization is when the project believes the teams will take care of all testing and coordination themselves. This does not happen because each team is only responsible for their own planning,

design and test. Anything outside the team's area of development (features, functions or sub-system) may be forgotten, unless someone enforces the team to be part of the overall system design and system testing. If there is no one appointed to lead these activities they may be missed (or done with poor results).

The role of the Scrum Master includes to coach the team and facilitate the sprint planning. Then also make sure the teams communicate with each other. There should be one full time Scrum Master per team (when team size is 5 – 9 persons) to be able to support all aspects needed regarding communication and coordination.

In addition to the Scrum Master, a system design coordinator and a system test coordinator may be needed. The system design coordinator leads the system design (act as the lead architect of the system). The system test coordinator plans and coordinates all the system testing activities and the system test team (if there is a separate system test team in the project).

**Test coordination** may include the following activities:

- > To plan, control, coordinate and follow up on all test activities that span across the teams. It may include any end-to-end or scenario tests with features developed in several teams. It could also be any characteristics or other system level tests (e.g. performance, stability, installation, security, reliability, integrity, etc).

The test coordinator ensures common test activities are included in each teams sprint planning. The test coordinator participates in the Scrum of Scrum meetings or separate test coordination meetings to follow up on the system test and cross functional testing.

- > To prepare the system anatomy showing the dependencies among the features in the system. The system anatomy should be prepared together with the system design coordinator (lead architect) and the person responsible for the configuration management in the project. The test coordinator should be part of the release planning to ensure all parts included are possible to test in an efficient and effective way. The integration points should be aligned with the system and cross functional test planning.

- > To analyze the User Stories regarding testability and completeness. From the analysis an overall test strategy and plan should be prepared in cooperation with the teams. The purpose of the plan is to give an overall picture of the system level/cross functional testing to ensure the test environment, tools and other parts needed outside the scope of each teams sprint planning are covered. A normal sprint plan which only covers one or two sprints ahead is not enough

when a system test environment is to be ordered/built, which may include prototype hardware (can take long time to deliver) or other equipment, installation and configuration.

- > To assure any purchase of test tools, test environments or any other parts needed for testing is aligned to get a good price from the tool vendors and a common test configuration. Otherwise it will be difficult/expensive to maintain if all teams have their own tools and environments.

TWEETABLE



**During the release planning stage, you should start to define the overall testing goals, strategies and follow up procedures to guide the project and the team in the right direction towards quality.**

TWEETABLE



**After the sprint planning meeting and your initial sprint test planning, you need to be more specific about how you are going to test what has been included in the Sprint.**



## 3.3 Sprint Iterations

Detailed test planning needs to be done for each Sprint. The overall test plan and strategy should be used as a guide to make the detailed planning easier.

Identify the test goals and what should be included from the back log, and plan all testing tasks accordingly.

Consider the following in the **detailed test planning**:

### What

Should be tested/not tested? What is the scope and goal of this Sprints tests? Which test aspects should be included? Are the User Stories testable? Are they complete enough (any vital information missing)?

### Why

Should the selected areas be tested/included and why should certain parts be excluded?

### Who

Will be in charge of testing of the different parts and aspects of the system?

### Where

In which test environment? Is there any need to change the environment in any way? What is the required test configuration?

### When

Can the different testing tasks be started in the Sprint and how often can/should they be performed (e.g. regression tests)?

### How

What test approach do we need? Which test methods, techniques, tools and test data should be used and how? How and what should be automated?

### Dependencies

What are the relationships and dependencies

to other functions, code units, system parts, third party products, technology, tools, activities, tasks, teams, test types/levels and constraints (e.g. Configuration Management, Product Management, used program libraries, databases, etc.)?

### Risks

Which areas have the most and the least risks? What testing problems may occur?

### Prio

Which parts need to have the highest priority of being tested regarding importance to the customer/users and the associated risks?

### Time

How many hours of testing work is available in the team and how much would be needed considering the User Stories planned for the Sprint and the related testing tasks?

At the end of the Sprint a retrospective meeting should always be held to give feedback to the current Sprint planning regarding planning accuracy, contents and the efficiency of the work in the team (check the velocity prediction versus the actual figures of finished tasks and features during the course of the Sprint).

It is also very important to report all found defects during the Sprint to keep track of what faults are discovered during testing [Dubakov -10]. Even if the faults are corrected the next day in the same Sprint. The defects should be noted in a defect catalog to be included in the used taxonomy of common faults. The taxonomy is one of the main sources for the selection where to focus current test effort from a risk perspective. Common types of faults have a high probability of being repeated and show up again.

Tool support is vital to make the testing work efficiently. List the tools you need for all testing tasks and identify how to get them. Assure you have enough knowledge and

support from the (third party) tool vendors where you purchased or downloaded the tools (if you use freeware).

Example of sources where to find useful tools for Agile testing:

<b>Open Source Testing Tools</b>	<a href="http://www.opensourcetesting.org/">http://www.opensourcetesting.org/</a>
<b>A comprehensive list of testing tools</b>	<a href="http://www.aptest.com/resources.html">http://www.aptest.com/resources.html</a>
<b>Combinatorial testing tools</b>	<b>Allpairs:</b> <a href="http://www.satisfice.com/tools.shtml">http://www.satisfice.com/tools.shtml</a> <b>PICT:</b> <a href="http://www.pairwise.org/tools.asp">http://www.pairwise.org/tools.asp</a>
<b>Stress test of input fields</b>	<b>Perlclip:</b> <a href="http://www.satisfice.com/tools.shtml">http://www.satisfice.com/tools.shtml</a>
<b>Logging tools</b>	<b>SpectorPro:</b> <a href="http://www.spectorsoft.com/">http://www.spectorsoft.com/</a> <b>HttpWatch, Ruby And Watir:</b> <a href="http://www.httpwatch.com/rubywatir/?gclid=COXolMerr6QCFY0_3godmVG70A">http://www.httpwatch.com/rubywatir/ ?gclid=COXolMerr6QCFY0_3godmVG70A</a> <b>BB TestAssistant:</b> <a href="http://www.bbsoftware.co.uk/BBTestAssistant.aspx">http://www.bbsoftware.co.uk/BBTestAssistant.aspx</a>
<b>Test management</b>	<b>Testexplorer:</b> <a href="http://www.sirius-sqa.com/products.shtml">http://www.sirius-sqa.com/products.shtml</a> <b>Session Tester:</b> <a href="http://sessiontester.openqa.org/">http://sessiontester.openqa.org/</a> <b>(Testexplorer also have built in logging functions)</b>
<b>Resource viewer</b>	<b>Viewing of resources in executable files (error messages or other text messages):</b> <a href="http://www.glocksoft.com/resource_viewer.htm">http://www.glocksoft.com/resource_viewer.htm</a>
<b>Test automation</b>	<b>Key Word Driven test automation:</b> Rasta: <a href="http://rasta.rubyforge.org/index.html">http://rasta.rubyforge.org/index.html</a> <b>Robot Framework:</b> <a href="http://code.google.com/p/robotframework/">http://code.google.com/p/robotframework/</a> <a href="http://www.onestoptesting.com/automationframework/keyword-driven-testing/">http://www.onestoptesting.com/ automationframework/keyword-driven-testing/</a>

For characteristics, specialized tools may be needed for a more thorough testing, which are usually recommended and used by the experts in their various fields of expertise.

#### Estimation

The general approach to estimating in agile development is based on the experience of the team by using functional analysis. The following description of how to estimate in Scrum is based on [Cohn -05].

The relative size of a User Story is calculated in so called Story Points. It is used to measure the effort required to implement a story (including all testing required). The number tells the team how much effort the story takes to develop. Each story should be possible to fit into a Sprint.

A typical story point range could be: 1,2,4,8,16 or X Small, Small, Medium, Large, Extra Large.

Story Points are relative and do not directly co-relate to actual hours [Cohn -10]. Since Story Points have no relevance to actual hours, it makes it easier for the Scrum teams to think abstractly about the effort required to complete a story.

A method that is used in agile development is called planning poker [Poker -10]. Planning poker is a consensus-based estimation technique for estimating effort or relative size of tasks based on the Delphi method [Delphi -10].

When estimating, don't forget to include the characteristics the system needs to fulfill. If you have missed specifying the characteristics in User Stories, you need to point it out and consider how it should be applied for the User Stories you already have (or create new ones). For example, how much effort would it take to assure the performance of the system results, in response times of less than one second, when the user interacts with the different features in the system?

Participants in planning poker includes everyone in the team (programmers, testers, database engineers, analysts, user interaction designers, and so on). The number of planning participants should not exceed ten people (otherwise split the planning session into two or more).

At the start of planning poker, each estimator is given a deck of cards. The cards are numbered in the following way: 0, 1, 2, 3, 5, 8, 13, 20, 40, and 100.

The moderator for the planning session reads the description for each User Story to be estimated. The moderator is usually the product owner or a system analyst. The product owner may answer any questions during the meeting for clarification. The goal in planning poker is to derive an estimate that is good enough with a minimal amount of effort.

Each estimator selects a card representing his or her estimate. Cards are not shown until each estimator has made a selection. All cards are simultaneously turned over.

The estimates may differ significantly in the first round. When they differ, the high and low estimators should explain their estimates. They may have different solutions in their mind when thinking about how much effort is required.

The group can discuss the User Story and the estimates for a few more minutes. The moderator can take any notes that may be helpful when programming and testing later on.

After the discussion, each participant makes a new estimation by selecting a card. Usually the estimates will start to converge by the second round. If they have not, continue repeating the process. The goal is to agree on an estimate that can be used for the development of the story.

When everyone is involved in the estimation the accuracy will increase, since more discussions and elaboration of possible solutions are made (compared to if only one person did the planning and estimation).

With many small User Stories it may be more practical to group them together instead of estimating each one separately.

## The Risk Analysis

The risk level of the software and system is one of the most important factors regarding how much test effort is needed. The following factors may be considered. The list is based on [Schaefer -05]:

1. Identify and analyze missing, ambiguous or vague parts of the specifications (User Stories) for the system.

2. Analyze the use of the system within its overall environment. Analyze the ways the system might fail. Analyze possible costs and consequences of the different failure modes.
3. Identify areas where many users will see and experience a failure if something goes wrong. Also consider how tolerable the user is to the failure.
4. Identify the functions that may be used frequently, and the functions used only a few times. Some functions may be used by many users and some by few.
5. Identify the areas of the system critical for the business processes and the business for the user/customer. Also consider the costs of failure and the usage frequency of the business critical parts.
6. Identify any complex areas of the implementation that may include large modules, many variables in use, complex logic, large data flows, and a complex functional behavior.
7. Identify the changes per functional area (features) and per structural area (units). Identify the areas that had exceptionally high changes (check the identifier for the revision of the units). Many changes may indicate design with insufficient analysis and implementation (or it might be a very central part of the system that is subject to frequent changes because of a high level of coupling and a low level of cohesion).
8. Identify areas using new technology, solutions or methods.
9. Identify the number of people involved in the design and implementation of the different areas of the system. The more people on the same task, the larger the overhead is for the communication, the chance increase that things go wrong.
10. Identify areas where the developers have been replaced during the course of the project.
11. Identify areas of the system that have been developed by designers/teams with a long distance between each other. The level of communication between developers decreases by distance.
12. Identify areas of the system that have been developed under high time pressure, where possible shortcuts have been made during the analysis, design and unit test phase.
13. Identify areas of the design and implementation that have been optimized. Possible shortcuts or insufficient analysis and unit tests of the optimized area may lead to problems.
14. Identify areas where many defects have been found in the previous project and in operation. There is a high chance that defect prone areas persist unless they are completely redesigned. Look at the trouble report descriptions to find common causes of the problems. Make a prioritized list (instability checklist) of causes based on the severity of the problems.
15. Identify problem areas already uncovered in the inspections of the requirements, during design or unit test

**Coupling:** *Two components are loosely coupled, when changes in one never or rarely necessitate a change in the other.*

**Cohesion:** *A component exhibits high cohesion when all its functions/methods are strongly related in terms of function.*

(depending on when the risk analysis is performed in the project lifecycle).

16. Identify to what extent the system and its functions have been used in a previous release. Completely new functional areas are most defect prone.

The risk table - Example

<b>Probability</b> <b>Consequence</b>	<b>Incredible (1)</b>	<b>Improbable (2)</b>	<b>Remote (3)</b>	<b>Occasional (4)</b>	<b>Probable (5)</b>	<b>Frequent (6)</b>
<b>Significant (4)</b>	Negligible	Tolerable	Undesirable	Intolerable	Intolerable	Intolerable
<b>Major (3)</b>	Negligible	Tolerable	Undesirable	Undesirable	Intolerable	Intolerable
<b>Minor (2)</b>	Negligible	Negligible	Tolerable	Undesirable	Undesirable	Intolerable
<b>Insignificant (1)</b>	Negligible	Negligible	Negligible	Tolerable	Tolerable	Undesirable

<b>Probability</b>	6	<b>Frequent</b>	The risk is likely to occur frequently. The risk will be continually experienced (daily or more often).
	5	<b>Probable</b>	The risk will occur several times. The risk will be expected to occur often (once per month).
	4	<b>Occasional</b>	The risk is likely to occur several times (once per year).
	3	<b>Remote</b>	The risk is likely to occur sometime in the lifecycle of the product (once per 10 years).
	2	<b>Improbable</b>	The risk is unlikely to occur but possible (once per 100 years).
	1	<b>Incredible</b>	The risk is extremely unlikely to occur (once per 1000 years).

<b>Consequence</b>	4	<b>Significant</b>	Immobilizing failure that prevents train movement or causes a delay to service greater than a specified time and/or generates a cost greater than a specified level.
	3	<b>Major</b>	A service failure that: - must be rectified for the system to achieve its specified performance and - does not cause a delay or cost greater than the minimum threshold specified for a significant failure
	2	<b>Minor</b>	A failure that does not prevent a system achieving its specified performance.
	1	<b>Insignificant</b>	A failure of no significance regarding intended usage or characteristics.

Regarding the consequences in the above example, there might be more critical levels than what is shown in the table, e.g. catastrophic consequences, but this example is just to show the principles and methods of doing the risk assessment and how to present the results.



TWEETABLE

The risk level of the software and system is one of the most important factors regarding how much test effort is needed.

To determine the testability of each risk might be beneficial to see if it is possible to use any known methods or techniques to

evaluate it or not in testing (to a reasonable cost). Otherwise the team needs to focus on other mitigation strategies instead.


<b>Testability</b>	4	<b>Impossible</b>	No known method exists to evaluate the risk, or it is far too expensive/difficult to set up and perform a test.
	3	<b>Hard</b>	It is difficult to test. It requires a lot of resources to set up and do the test and it takes considerable time.
	2	<b>Moderate</b>	It requires a reasonable amount of resources, time and effort to do the test, but it is doable with available resources.
	1	<b>Easy</b>	The risk requires only a small amount of time and resources to be evaluated.

<b>Priority Probability * Consequence</b>	1	<b>Intolerable</b>	Shall be eliminated. Must be tested/evaluated.
	2	<b>Undesirable</b>	Shall only be acceptable when risk reduction is impractical and with an agreement with the customer.
	3	<b>Tolerable</b>	Acceptable with adequate control and with the agreement of the customer.
	4	<b>Negligible</b>	Acceptable without any agreements.

The above example is from a safety critical application (train signalling system). In your case you may want to simplify it to suit your needs of assessment, in case you develop less critical applications. It is however always a good idea to learn from how they do in more critical applications, when making risk


assessments. You can always get inspired and find good ideas you may incorporate in your own process (since they have probably spent a significant amount of time thinking about how and what to look for regarding possible problems in their systems).

TWEETABLE



Since software testing primarily is a learning and evaluation process, you need to look at what your reasoning structure looks like.

TWEETABLE



To find good ideas about what to test is to ask yourself (and your colleagues): What is most important to find out about this system?



When doing Risk Based Testing [Claesson1-02] to explore the risks, it might look something like this (the example is taken from testing an Axle Counting system in a train signaling system):

**Requirement:** In release 2 the system shall support train speeds up to 160 km/h. The axle counter system shall only need to support 160 km/h for wheel sizes down to 335 mm diameter and 75 km/h down to 318 mm diameter, all operating on rail sections UIC54 or larger.

Prio	Risk description	Probability * consequence	Actions
1	<p><b>Risk_1: Wrong values regarding number of passed axles received from the axle counting system.</b></p> <p><b>Cause:</b> Incomplete requirements. Unknown behavior of the system outside specified boundaries.</p> <p>Incorrect system boundaries implemented regarding acceptable wheel size and speed for the axle counting to be accurate.</p> <p><b>Effect:</b> The consequence may be that the signaling system consider a train too short (or long) when passing with a speed over 160 Km/h and/or a wheel size larger than 335 mm.</p> <p>Serious failure modes may also occur when the summarized incorrect axle counter values make the interlocking system believe that a track section is unoccupied when it is actually occupied and vice versa.</p>	<p><math>4 * 4 = 16</math></p> <p>Testability = 3</p>	<p><b>Risk reduction and evaluation strategy:</b> Perform a requirements validation with the customer and/or the product management.</p> <p>Perform a lot of boundary, domain and negative Test Cases to evaluate the “real” limits of the axle counter system. Also consider which parameter values are the most common and which ones we know exists that exceed the specified limits.</p> <p><b>Test object:</b> <u>Axle counter test.</u></p> <p><b>Test cases:</b></p> <ul style="list-style-type: none"> <li>- Test of individual parameters and combinations thereof. Select Test Cases based on the boundaries indicated in the domain analysis.</li> <li>- Test outside of the parameter boundaries.</li> </ul>

Note that doing risk analyzes is not part of the “standard” Scrum methods in Agile development, since it is assumed the risks are taken care of “on the fly” by the team. But if you put in “doing a risk analysis” as a task in your Sprint plan, it can be fitted into the Scrum way of working and you will benefit a lot from looking at what problems you might run into in advance, so you are able to avoid them instead of realizing the consequences and having to correct it later on (which may slow down the progress significantly).

One example of a costly risk is performance problems. If you have developed a .NET solution when creating a Client/Server system and start to experience slow response times when accessing the database, it may be an indication that you have created an inefficient architecture with costly solutions when accessing the database (costly for the system performance resulting in a high load for the processor in the application server).

Further examples of risk situations to be aware of in a Client/Server system:

Risks	Mitigation/Solution ideas
<p><b>Slow response times when many users try to access the same database tables.</b></p> <p><b>Cause:</b> Too high database isolation level. The designer may have chosen this solution to avoid that:</p> <p style="padding-left: 20px;">Data is accessed/updated by multiple users at the same time.</p> <p style="padding-left: 20px;">Transition fails to perform all necessary actions due to a crash.</p> <p>It is much easier for the programmer to implement a solution with a high isolation level than a low.</p> <p><b>Effect:</b> The higher the isolation level is, the slower the system gets because the database table is locked (when using the highest level) until the current user is finished accessing it. If there are many others trying to access the same database table they have to wait.</p>	<p>Lower the isolation level by including efficient rollback mechanisms in the database access code.</p> <p>Update your programming guidelines how to design your application to make the database access more efficient and effective.</p> <p>Measure access times for each type of transaction to the database.</p> <p>Perform load and stress tests on the whole application with various database contents (size and complexity). Use different user profiles and server configurations with both a “normal” use of the application, as well as a number of “worst cases”.</p>

As you can see from the previous examples, the risk assessments will have a major impact on how to make an efficient and effective test strategy.

At the start of each Sprint, you, as a tester, will also be part of the Sprint planning made in your team. The test planning part should be done in parallel with the Sprint planning.

The purpose of the Sprint planning is: **To decide what to deliver in the next Sprint**

The following activities are included in the planning meeting. The list is based on [Kniberg -07]:

- > Clarify User Stories if needed.
- > Define the definition of "done" (DoD) for the Sprint.
- > Define how the Sprint demo should be performed.
- > Break apart or combine User Stories.
- > Estimate time.

- > Update priorities in the product backlog.
- > Calculate resource availability.
- > Decide which User Stories to include in the Sprint.
- > Analyze risks.

#### Key responsibilities:

Team: Time estimation and capability of the team.

Product owner: Prioritization.

The testers contribute in all parts of the Sprint planning and assure all of the testing related tasks are planned for and included in the Sprint backlog.

After the sprint planning meeting and your initial sprint test planning, you need to be more specific **how** you are going to test what has been included in the Sprint.

Go back to your high level test planning previously made and take a look for ideas.

Below is a list of additional test methods you may want to consider when testing in the current Sprint:

**1) Test in pairs.**

Two team members are sitting behind one machine while testing. One team member is in control of the mouse and keyboard, the other one is making notes, discussing the test scenarios and asking questions. One of the two should be a tester and the other one could be a developer or business analyst (or another tester).

**2) Prepare test charters in advance.**

A test charter is a description of how an end user is intended to use the system in a real situation, by executing specified tasks in different ways. The intention is to execute what may be useful for him/her or another actor, which may show weaknesses in the way it is intended to be used, considering what the user wants, needs and expects.

**3) Use exploratory testing.**

Exploratory testing seeks to find out how the software actually works, and to ask questions about how it will handle both difficult and easy situations. The quality of the testing is dependent on the tester's skill of creating tests and finding defects. While the software is being tested, the tester learns things that together with experience and creativity generate new good tests to run. Exploratory testing has always been performed by skilled testers [Bach2 -01], [Bach3 -03].

**4) Build tests incrementally.**

Incremental test design is when test cases/charters are gradually built from a basis of User Stories or other input. Test cases are incrementally made starting from a simple case to more complex tests.

**5) Use test design patterns.**

A test pattern could be a template used for the automatic generation of component tests or guidelines for which types of functional tests might be useful. Test patterns can be used as test ideas.

**6) Perform keyword/data driven tests.**

Key-word driven test is a form of data driven test automation, where the test logic is hidden in procedures made by skilled script programmers. A list of data values are fed to repeated invocations of a test script. The test cases are usually made in Excel and are easy to change and maintain.

**7) Perform end-to-end testing.**

End-to-end testing means to test a complete application in a "customer like" environment that mimics a real-world use (usually scenario based with as complete and real scenarios as possible) [Lambert -10]. That could include activities where users interacting with a database, using network communications, or interacting with other hardware, applications or systems.

**8) Use scenario based testing.**

Scenario testing is a specification based test design technique in which test cases are designed to execute user scenarios. The scenarios could be based on the underlying business process in which your application will operate, and hopefully solve the problem that the customer and user wants to solve with your system, in an efficient and effective way.

**9) Use automation for test data generation and execution.**

Tools can be used to generate large amounts of test data (like the tool Perlclip), or to generate test cases for combinatorial testing. Test automation

is also beneficial for regression testing to assure a new build from the developers work (basic functionality, also called smoke tests) and that previously corrected faults have not come back because of problems with your CM-process where old versions of some modules reappear again.

#### 10) Perform frequent regression testing.

As mentioned above, regression testing is an important activity to ensure that a previously tested program, which has been modified due to functional changes or defect corrections, has not introduced any side effects (new bugs), as a result of the changes made. Regression tests should be performed as soon as the software or its environment is changed.

#### 11) Test all documentation.

One definition of documentation is “Any written or pictorial information describing, defining, specifying, reporting, or certifying activities, requirements, procedures, or results”. Documentation is as important to a product’s success as the product itself, and therefore needs to be tested together with the system.

#### 12) Log everything you do.

Logging is especially important when performing exploratory testing. Test ideas and the concurrent test design can be backtracked when defects are found and the test coverage can also be determined.

When you write incident reports include the recorded sequence when the fault occurred, as an attachment. In that way you do not need to describe so much on what happened and the designer can see for himself what you did.

Before you start testing it is also a good idea to ask your designers which system log files and system variable readings they would need, to be able to

solve the problem faster. Otherwise they may need to come back to you and ask for more information which you might not be able to give to them anymore because the state of the system changed since the problem was discovered and critical system variables have been overwritten by the continuing events in the system.

Now it is time to gather the team to find good test ideas for the functions and characteristics included in the current Sprint. This is the most difficult part, regardless of development method, what to test within a very limited time frame, limited test environment, limited data and configuration/setup compared to what could happen in real life system operation.

Nowadays there are tools that support this process. By using the tool Session Tester (mentioned earlier in the tools section), both test questions and test ideas can be written and linked to each other. Session Tester is an exploratory testing tool for managing and recording Session-Based Testing. It has a timer so you limit your test sessions to a particular length, and it provides an easy way to record session notes [<http://sessiontester.openqa.org/about.html>].

To find good ideas about what to test is to ask yourself (and your colleagues): **What is most important to find out about this system?** (Based on previous test analyzes where you hopefully got a good understanding about what the system is supposed to do).

Example of questions to start off your brainstorming session:

- 1) What happens if .....
- 2) What should happen when .....
- 3) Will the system be able to fulfill all its requirements?
- 4) What are the expectations and needs from the customer?

- 5) In what way may the system fail?
- 6) What problems were found in the previous release?
- 7) Are the requirements and the input specifications possible to understand and test?
- 8) Will the system be reliable and resist failure in all situations?
- 9) Will the system be safe in all configurations and situations it will be used?
- 10) How easy is it for real users to use the system?
- 11) How fast and responsive is the system?
- 12) Is it easy to install (and configure) onto its target platform?
- 13) How well does it work with external components and configurations?
- 14) How effectively can the system be tested (e.g. can log files be read)?
- 15) Which are the different operating environments the system will be used in?

A general checklist, as the one above, can be used as a start. Continue by filling in what is important to find out about your system. Catch the test ideas early so they can be elaborated during the project.

Discuss and list your test ideas in the team. Transfer the ideas into test charters just in time before the test execution starts. Otherwise there is a risk of specifying test ideas that are no longer relevant because of recent changes in the system, its design and the resulting behavior.

A test idea represents a question you want

to ask to explore a component or system under test to find out how it really works and if its behavior may be considered correct/acceptable or not. The point of running the test is to gain information. The tests may or may not be specified in great detail, which is ok, as long as it is clear what the idea of the test is and how to apply that idea to some specific aspect of the product.

Tests can be of the following categories:

#### Normal case

- Normal flow of events. Also called a positive test.
- Normal configuration (according to common usage profiles).

#### Negative case

- Negative test to show that the system can handle failure conditions.

#### Interaction/ combination case

- Test case to show how the function or system interacts with other functions, SW, HW or systems.

#### Scenario

- A sequence of tests based on Requirements, Operational Scenarios or User Stories. A scenario could be a sequence of one or more User Stories that together become an end-to-end test.

#### Performance

- Volume and stress test for specific configurations and scenarios.

#### Usability

- Navigation and screen presentation, command structure, goal oriented based on user manuals.

#### Other Characteristics

- Consider the customer value, analyze which other important characteristics to cover. See also the standard ISO 9126 for important characteristics of a typical software system.



This is an example of the contents of a Test Charter (as suggested in this paper). A Test Charter consists of test objectives, and test ideas used for exploratory testing [Bach3 -03].

### Actor

Type of user.

The role, you as a tester, are going to play when using the system. If you test characteristics you may need to have multiple roles.

### Purpose

Describe the function, web page, test idea, etc. to be tested.

The purpose represents the theme of the charter including what particular objective the actor wants to achieve.

### Setup

Preconditions i.e. concerning HW, content of data base(s), etc.

What needs to be in place in order to start using the intended function, i.e. the preconditions to be able to test (configuration and the initial system state, e.g. a start menu should appear on the screen with login fields).

### Priority

The importance of these tests.

High/low, based on what the priority of the requirements is, or other considerations (risk level).

### Reference

Specifications (e.g. Requirements Specification), risks or other information sources.

### Data

Whatever data is needed to perform the described activities.

References to data files with input data that is needed, e.g. an Excel sheet. Variations of data to be made, based on the initial data can also be specified, e.g. use boundary values of valid and invalid input data for all parameters. Use pair-wise techniques to generate input data to catch combinatorial problems.

### Activities

A list of possible actions and test ideas.

Ideas of what the actor may want to do with the system, e.g. "Log on to the system as a super user", "Add a new user", "Attempt to delete a non existing user". Also include negative cases.

### Oracle notes

How to evaluate the product to determine correct results.

Things to pay special attention to regarding system response and what is expected as an adequate/correct outcome, e.g. to check that correct error messages are given, that informs the user about what mistakes they made, and how to do it correctly next time. It could also be a reminder to the tester to check for buffer consumption or other important resource usage.

### Variations

Alternative actions and evaluations.

Additional system or user events that could occur at any time, when the normal activities are performed as described above under Activities, e.g. another super user logs on to the system trying to manipulate the same data as the first is working on, or the network connection suddenly fails.



— TWEETABLE —

**Agile testing can be divided into four different quadrants to show different aspects of testing and purpose.**



— TWEETABLE —

**Testing should stop when the probability of remaining faults is reduced to a level that can be accepted by the customer/user [Beizer -90].**



## Example of a Test Charter

### Actor

Normal user.

### Purpose

To evaluate if the copy/paste function of pictures works in our word processor together with the most commonly used word processors on the market (Word, Power Point., etc.) and other programs where pictures can be inserted in the copy buffer for copy/paste operations in the PC. The purpose is also to see that no information is lost or corrupted.

### Setup

A Dell PC with 2 Gb memory, “our” word processor to be tested, the Microsoft Office package professional and the home edition 2003 patch level xx, PDF reader version 7.1, Notepad version 4, Internet Explorer version 7, Opera version 5, Mozilla.....etc. (the setup might be common for several Charters and can therefore be described and referred to instead of repeating the same information in every charter).

### Priority

High, because this function is used very frequently both within our own word processor, but also between other word processors and programs, the user may want to copy/paste pictures with ours.

### Reference

Requirement abc\_1 in document Def-09:034 revision R1.1. Risk number 3 from the risk assessment held on April 4 2010 regarding the copy function, which is documented and followed up in the document Rsk-09:012.

### Data

A variety of pictures with different resolutions, both vector graphics as well as bit mapped pictures. The pictures could be photos or figures in web browsers for example. Complex pictures are also included and pictures which might be copy protected in some way.

### Activities:

1. Copy/paste a picture in our word processor from one location to another in the same document.
2. Copy/paste a picture to and from our word processor into/from Word, PowerPoint and Excel.
3. Copy a picture into our word processor from a variety of the mostly used Web browsers.
4. Copy/paste a picture to/from the most commonly used web site building tools such as Dreamweaver from Adobe.
5. Copy a picture from a PDF document using Acrobat Reader into our word processor.
6. Try to copy a write protected picture from the web or other source into our word processor.
7. Try to copy/paste a variety of non readable and some readable ASCII-characters or corrupted pictures.

### Oracle notes

- > Look if the size of the pasted picture changed on the screen (it should not).
- > Check if there is any loss in the resolution of the picture especially when you copy and paste from other programs to or from ours.
- > Check which is the highest resolution of a picture that can be copied, and how that affects the system.
- > Check for memory leaks and how much memory the copy/paste operation takes from the system and how that affects the use of other programs. Other programs should not slow down or be affected in any way.
- > Print pasted pictures to see if there are any differences in color, resolution or any other anomaly.

### Variations:

- Try out (and find) the boundaries of how large pictures are possible to copy/paste.
- Perform copy/paste with a large number of items in the copy/paste buffer in your PC.

- Try to fill the copy/paste buffer to its limit and then copy/paste a picture and see what happens.
- Try the longest file name of the picture you can type before making the copy/paste. You may use the tool Perlclip (download from <http://www.satisfice.com/tools/perlclip.zip>) to generate file names with a million characters or more if you want.

A typical test charter includes the work of a pair of testers working together for a 90 minute period of uninterrupted time, i.e. time boxing (cell phones shut off, no email pop-ups active, etc.).

The test charter should suggest what to be tested, how it should be tested and what problems to look for (OBS, not a detailed step-by-step instruction).

Chartered testing is mainly used for exploratory testing and requires more system and testing knowledge than scripted testing (but the fault finding efficiency is 2 - 4 times higher). Try to play chess with someone using a predefined strategy without considering changing the strategy, based on what moves your opponent makes.

Chartered testing encourages testers to react on the response from the system, and to think for themselves what a correct outcome should look like, and also use their creativity on how to break the system. The tester will also be encouraged to look for different ways on how it might fail to meet the user's needs and requirements (e.g. usability aspects).

When testing, make sure you cover the following. The list below is based on [Bach4 -10] and [Bach6 -10]:

### Structure

Everything that comprises the physical product.

System parts, sub-systems, interfaces between sub-systems, etc.

### Functions

Everything that the product does.

User interface, system interface, application, calculations, timerelated, transformations/modify data format, startup/shutdown, multimedia, error handling, interactions, testability/log files.

### Data

Everything that the product processes.

Input, output, preset data/default values, persistent data stored internally, sequences - ordering or permutation of data, variations in the size and aggregation of data, noise - any data or state that is invalid, transformations over the lifetime of a data entity.

### Platform

Everything on which the product depends (and that is outside your project). External hardware, external software, internal components – that is embedded in your product but is produced outside your project (e.g. third party products).

### Operations

How the product will be used.

Attributes of different users, operational environment, common use, disfavored use, extreme use.

### Time

Any relationship between the product and time.

Delays of input/output, response times under different conditions and use, changing rates - spikes, bursts, hangs, bottlenecks, interruptions, concurrency.

Some further ideas of how to expand suggestions of the variations in the Test Charter, based on [Bach2 -01], [Bach3 -03], [Bach4 -10]:

### Data usage/change

Different users try to manipulate the same data simultaneously.

### Interruptions, aborts

Unfinished activities are common in work

environments with backtracking and a lot of distractions.

### Object lifecycle

Create some entity, then change it, delete it and create the same entity again (and change and delete).

### Long period activities

Transactions that take a long time to execute, periodic events and maintenance activities.

### Function interactions

Related features operating on the same data or being part of complex operations.

### Learning curve

Do things a novice user may do with your system and see how easy/difficult it is to complete a task.

### User mistakes

Make realistic mistakes in similar ways that distracted and busy people may do.

### Complex data

Use complex user data in ways that may cause the system to crash.

When executing your Test Charters it is beneficial to use a structured approach. A well tested method is the scientific approach. Scientists have been using the same method of research for more than 100 years. This is applicable and can be used in our case too.

Identify the specific area to be explored according to your Test Charter.

Use a “scientific” approach to explore the different parts in the system [Claesson2 -02].

- 1) Observe and decide which elements to test first.
- 2) State questions (what happens if.... Speculate about possible quality problems).

3) Form hypothesis (this might happen.... when..).

4) Design the experiment (i.e. use one of the test ideas in the charter or come up with your own based on your previous observations).

5) Test the hypothesis (execute your test idea).

6) Draw conclusions (compare the outcome with your test oracle).

7) State additional questions (i.e. add test ideas until the issue has been resolved and you are satisfied with the obtained test coverage).

The most common task of testing is to find other people's mistakes, to find an efficient method, how to discover the mistakes, and to present the results, which may be used as a basis for decisions (of bug fixing, more testing, to determine risks or whether to release or not). Therefore social science methods may be useful as a method for the investigation about the system, the people creating and using it, and its behavior [Kaner2 -04]. Sometimes you may even start without any hypothesis and let them emerge from the findings about the system while testing [Edgren -10].

Don't expect that requirements and specifications always exist before you prepare a test. If your reasoning is “no specification – no test!” it will be counterproductive because it is the quality perceived by the customer that counts in the end. The user doesn't care if there were perfect requirements to begin with or not, he/she just wants a product that works and fulfills his/her needs.

When testing the system there are some key challenges to be aware about, and plan for how to handle. The following is based on [Kaner1 -04]:

### Learning

How to get to know the system under test. That includes to learn about how the product can be used, the market, the intended users, customers, which ways the product can fail, weaknesses about the product, how to test the product efficiently.

### Visibility

Which information is visible in the user interface, system log files or any other means of viewable information storage? What are the current and past states of the system?

### Control

Can all features and functions in the system be controlled, from any external or internal interface of the system, that is possible to give input (data) to?

### Selection

Which tests are the best, most efficient ones to run, in order to find the most important bugs?

### Execution

What is the most efficient way to run the tests?

### Logistics

What environment is needed to support the test execution?

### Oracle

How do you tell if a test result is correct or not (the oracle problem)?

### Reporting

How to replicate a failure and report it effectively?

### Documentation

What test documentation do we need?

### Measurement

What metrics are usable and applicable for our testing?

### Stopping

How to decide when to stop testing?

Also consider the following when observing the system behavior:

- > Initial, actual and expected system state.
- > Configuration and system resources.
- > Input from other cooperating processes, clients or servers.
- > Sequence of actions.
- > Navigation and usability.
- > Output format and contents.
- > Consistency.
- > Size, calculations, range.
- > Timing.
- > Relations.
- > Patterns.
- > Resulting system state after your test is finished, including uninspected outputs.
- > Impacts on connected devices/system resources.
- > Output to other cooperating processes, clients or servers.

A system can fail in many ways. Inattentional blindness is a common phenomenon where humans (often) don't see what they don't pay attention to. Therefore it is important to monitor all important events in the system. Consider your test environment's level of controllability and observability, i.e. how much can you actually control and see what is going on when you test?

While testing, you need to log everything you do (see suggested tools earlier in the document), or at least know what steps you took while testing. Otherwise you cannot backtrack what you have done, especially

if you find a fault you need to report (use the time stamps in your logging tool). Also take notes regarding your observations in a testing logbook to show your reasoning while testing [Jo.Bach - 00].

Since software testing primarily is a learning and evaluation process you need to look at what your reasoning structure looks like. Most people have an ad-hoc way of reasoning. This doesn't work very well when testing complex systems where you need to have both a backward and forward looking ability in several steps, as well as a way to analyze what you see and how you are going to act in the following steps in your evaluation/testing of the system.

To help, you may use a reasoning map to enhance your ability to sort information and make better test decisions. With some practice you can become a very skilled tester in a quite short amount of time.

## Thinking and reasoning structures

### 1 - Information processing

#### 1.1 Locate and collect

Collect all information you can get for further questioning and evaluation to learn more about the product, problem, test item and/or test results.

#### 1.2 Sort

Arrange items in different sets by ordering the same types or categories.

#### 1.3 Classify

Classify the information into related areas of the product (features, functions, characteristics, etc.).

#### 1.4 Compare and contrast

Compare two things, how they are alike and contrast, to tell how they are different.

#### 1.5 Sequence

Identify if the logical order of the information is relevant.

### 1.6 Analyze parts - whole relationships

In this section you make models out of the object you analyze. For example, state diagrams, system anatomy maps, functional decomposition diagrams, system architecture and interfaces, usage scenarios, characteristics, etc.

## 2 - Enquiry

### 2.1 Test conclusions and improve ideas

Generating test ideas to prove or disprove what you think, how the system is supposed to work.

### 2.2 Ask relevant questions

Use Critical, creative, scientific and lateral thinking [Kaner2 -04], [Wikipedia2 -11], [DeBono -99].

### 2.3 Predict and anticipate

Based on your experiences ask yourself "in what way may the system fail?".

### 2.4 Define the problem

When you are going to build a system you need to know which customer and user problems the system should solve and why.

### 2.5 Plan and research

If you get a deeper understanding of the reason why the system is built, you may be able to provide a solution that is better than the customer was expecting. You may also be able to create better test ideas.

## 3 - Creativity

### 3.1 Apply imagination

Imagine yourself using the product you are going to test from a user's perspective.

### 3.2 Suggest hypotheses

Speculate in which ways your product can be used and in what context?

### 3.3 Look for innovative outcomes

Ideas about failure modes or any other normal/abnormal outcome.



### 3.4 Generate and extend ideas

List test ideas and elaborate on alternatives and different themes.

## 4 - Evaluation

### 4.1 Judge value

Would a “normal” user of your product consider the provided features usable and valuable, or not?

### 4.2 Evaluate information

Make customer reviews and look at the results. Evaluate the information you have about the system and what the customer considers important. Evaluate the outcome of your tests.

### 4.3 Have confidence in judgment

Trust your instincts and judgment that your tests and analyses are good enough (but be aware of your biases).

### 4.4 Develop criteria

Specify criteria for when to stop testing and when you are done with your testing tasks.

## 5 - Reasoning

### 5.1 Use precise language

This refers to the problem of human understanding with written or verbal expression of words.

### 5.2 Reasons for opinions and actions

Before taking action or making a decision it is wise to ask yourself why?

### 5.3 Explain your thoughts

Use a (scientific) reasoning structure that cannot easily be questioned or denied.

### 5.4 Judge and decide from reasons and/or evidence

Use supporting information from the results of your tests.

### 5.5 Infer and deduct

Assure your conclusions are based on commonly known and expressed premises. A premise is a proposition upon which an argument is based or from which a conclusion is drawn.

There is also a simpler model you can use called the Deming PDCA cycle when testing or planning your tests [Wikipedia3 -11].

#### Plan

Collect information, analyze, define goals and strategies, decide and plan actions.

#### Do

Prepare and execute the actions in your plan.

#### Check


Check and evaluate the results.

#### Act

If needed, change your goals, strategies and plans based on the results from the previous actions.


In each iteration of the PDCA cycle you gain valuable knowledge, which you can add to the answers of your original questions (e.g. have all major faults been found, have we performed enough testing?).

**TWEETABLE**



**When integrating the system, the order in which functions and features are developed is very important.**

**TWEETABLE**



**The test strategy should be created in an evolutionary way during the whole project.**



The result of your tests can be presented on a so called “lo-tech testing dashboard” (expanded from the original format created by James Bach [Bach5 -99]).

Test area	Initial Risk Level	Needed test effort	Current risk level	Current test effort	W 1	W 2	W 3	W 4	W 5	W 6	Q ass.	Comments
Area 1	Low	Low	Low	None				0	0	0	☹️	Feature(s) not yet delivered from design and integration. Definition of Done not fulfilled for functional testing. No testing possible.
Area 2	Medium	Medium	Low	High	1	1	1+	2	2	2+	😊	On track, no faults.
Area 3	High	High	High	Blocked			1	1	1+	1+	☹️	Crashes, IR12345
Area 4	High	High	Medium	Pause	1	1	1+	1+	1+	2	😐	IR1212 under investigation.
Area 5	Medium	Medium	Medium	High	1	2	2+	2+	3	3	😐	Configuration problems.

### Needed Test effort:

#### Low

A low risk feature with an overall low ranking on needed test coverage.

#### Medium

A medium risk feature with an overall medium ranking on needed test coverage.

#### High

A high or critical risk feature with an overall high ranking on needed test coverage.

### Current Test effort:

#### None

No testing planned so far.

#### Start

No testing yet, but expect to start soon.

#### Low

Regression or spot testing only; maintaining coverage.

#### Medium

Broader coverage of the main parts.

#### High

Focused testing effort; increasing coverage.

#### Pause

Temporarily ceased testing, though area is testable.

### Blocked

Can't continue testing, due to blocking problem.

### Ship

Going through final tests and is ready to ship to the customer (internal or external).

### Quality assessment:



We know of no problems in this area that threatens to stop shipment or interrupt testing, nor do we have any definite suspicions about any.



We know of problems that are possible showstoppers, or we suspect that there are important problems not yet discovered.



We know of problems in this area that definitely stop shipping or interrupt testing.

The Current Risk level is based on the results from all performed preventive actions and/or all performed tests at the current week when the progress report was written.

The dashboard aims at showing the test and quality status for a system under test, based on subjective conclusions for all test activities in a release/project.

The dashboard will give some information to

the traditional testing questions:

- “What is the status of testing”?
- “What are you doing today”?
- “When will you be finished”?
- “Why is it taking so long”?
- “Have you tested \_\_\_\_\_, yet?”

The Test Area should include specific functions, features, objects or characteristics that are testable and make sense to the customer. For example: File, Edit, Insert, View, Format, etc. Minimize any overlap between the areas and keep them within a reasonable number (not more than 30).

Use the comment field to explain anything colored red, or any non-green quality indicator. It could be Problem ID numbers, reasons for pausing, or delayed start, the nature of blocking problems or why an area is unstaffed.

A column stating obtained test coverage could also be added or included in the weekly status of the tested area:

- 0 We have no good information about this area.
- 1 Sanity Check: Major functions & simple data.
- 1+ More than sanity, but many functions not tested.
- 2 Common Cases: All functions touched; common & critical tests executed.
- 2+ Some data, state, or error coverage beyond level 2.
- 3 Corner Cases: Strong data, state, error, or stress testing.

Level 1 and 2 focus on functional requirements and capabilities: can this product work at all?

Level 2 may obtain a 50 – 90% code coverage.

Level 2 and 3 focus on information to judge performance, reliability, compatibility and other “ilities”: Will this product work under

realistic conditions of usage?

Level 3 and 3+ implies “if there were a bad bug in this area we would probably have seen it”.



— TWEETABLE —

**In addition to the Scrum Master, a system design coordinator and a system test coordinator may be needed.**

## 3.4 End Game

During the end game the final regression tests and the finishing parts of the characteristics tests are performed (characteristics should be tested during the previous sprints as well, as much as possible).

Testing can start when individual functions or system parts are stable and work without too much disturbances. This often happens late in the project when there is not much testing time left (or time to correct faults, which may even impact the architecture of the whole system).

For characteristics testing a so called Goal-Question-Metrics model may be used [GQM -10]. GQM may not capture everything that is important because many aspects of the characteristics evaluation are not measurable (e.g. usability, security). A scientific approach is also useful [Claesson2 -02].

GQM is a top-down approach to establish a goal-driven measurement system in software development. It is very useful when testing characteristics which often include some sort of measurements.

A GQM model is a hierarchical structure starting with a goal specifying the purpose (e.g. to improve the call handling process) of the measurement, the object to be measured (e.g. the current call handling process), the

issue to be measured (e.g. the call setup time), and the viewpoint (e.g. from the users' point of view) from which the measure is taken.

The goal is refined into several questions (e.g. What is the current call setup time?).

Each question is then refined into objective metrics. The same metric can be used in order to answer different questions under the same goal. Examples:

- Average call setup time.
- Standard deviation.
- % of cases outside of the upper limit.

Example from a telecom application:

### Goal

The system should be able to handle traffic with at least 1000 simultaneous users active in the system. If no requirement exists, make a reasonable assumption based on similar or older versions of the system currently in service.

### Questions

Is the system capable of handling 1000 simultaneous users even though there are no clear requirements related to system performance or any current systems in operation that comes close to the target value? What if the system restarts in the middle of a high load situation? What are the main problems previously observed that we would like to know more about?

### Metrics

Number of active simultaneous users in the system, in call attempts or already connected. Restart times for node, base station and system. IR statistics with RCA (Root Cause Analysis) on serious faults (show stoppers) both from testing in the previous project and from current field operation to date in the current project that may affect performance.

In this example we need to state all the variables affecting the outcome of the test.

Make a table to find Equivalence Classes, Boundary Values and Domains. Depending on the type of characteristic, determine what values to select based on the goal and the questions. Also make sure the main goal is in line with the test goals, product/customer needs, and the overall R&D and Business Goals for your company.

Performance testing is one of the most common and important characteristics tests applicable for all systems. Below is an example of how to practically conduct a simple test (which too many projects tend to "forget").

Example:

- 1) Start the test execution as soon as a stable release with enough functionality has been received.
- 2) Prepare the load generation tool.
- 3) Start the measurement program (tool).
- 4) Identify which data to be collected.
- 5) Set up the test environment (load the test data).
- 6) Start with multiple use of one scenario.
- 7) Check that the system is stable.
- 8) Add scenarios until a full usage profile is reached.
- 9) Stop the test when the load objectives have been achieved.
- 10) Stop the performance monitoring processes.
- 11) Collect the measurement results and compare with previous results and what was expected.

Example of other characteristics that you may want to pay attention to [Kaner1 -04]

(There are a lot more to consider also [ISO/IEC 9126-1], [Testeye -10], [Al-Qutaish -10], [Ganesh -10], [Wikipedia1 -11]), the following list is based on [Bach4 -10]:

### Capability

Can the system perform the required functions?

### Reliability

Will the system be able to resist failure in all required situations?

- > *Error handling*: The product resists failure in the case of errors, is graceful when it fails, and recovers within the time frame set by the Service Level Agreement.
- > *Data Integrity*: The data in the system is protected from loss or corruption.
- > *Safety*: The product will not fail in such a way as to harm life or property.

### Usability

How easy is it for a real user to use the product?

- > *Learnability*: The operation of the product can be rapidly mastered by the intended user.
- > *Operability*: The product can be operated with a minimum of effort.
- > *Accessibility*: The product meets relevant accessibility standards and works with O/S accessibility features.

### Security

How well is the product protected against unauthorized use or intrusion?

- > *Authentication*: The ways in which the system verifies that a user is who she says she is.
- > *Authorization*: The rights that are granted to authenticated users at varying privilege levels.
- > *Privacy*: The ways in which customer or employee data is protected from unauthorized people.
- > *Security holes*: The ways in which the system cannot enforce security (e.g. social engineering vulnerabilities).

### Scalability

How well does the deployment of the product scale up or down?

### Installability

How easily can it be installed onto its target platform(s)?

- > *System requirements*: Does the product recognize if some necessary component is missing or insufficient?
- > *Configuration*: What parts of the system are affected by installation? Where are files and resources stored?
- > *Uninstallation*: When the product is uninstalled, is it removed cleanly?
- > *Upgrades*: Can new modules or versions be added easily? Do they respect the existing configuration?

### Compatibility

How well does it work with external components & configurations?

- > *Application*: The product works in conjunction with other software products.
- > *Operating System*: The product works with a particular operating system.
- > *Hardware*: The product works with particular hardware components and configurations.
- > *Backward*: The products work with earlier versions of itself.
- > *Resource Usage*: The product doesn't use more memory, storage, or other system resources than necessary.

When testing is coming close to an end it is important to look at your current risk list again. The risks should be looked at in every sprint on a continuous basis to keep them up to date.

The purpose of the risk evaluation is to:

- > Evaluate the testing results against the current list of the most important risks.
- > Keep track of the most serious faults found.

- > Update the list of current system risks.
- > Change priority or put in new risks in the risk list as a result of the current test results.
- > Give input for changes of the test strategy if needed.
- > Give input for changes of the test case, or test idea selection methods.
- > Initiate additional test activities if needed.

The result of the risk evaluation should be used to:

- > Decide when to stop testing.
- > Inform the project and the customer of the current risks and the quality of the system.

It is also a good idea to analyze the faults you have found so far to be able to draw any conclusions about the current quality and any systematic problems that need to be addressed.

Example:

#### Divide the faults found

- Group faults into categories of how serious they are.
- Prioritize the faults according to severity and occurrence.
- Make a top 10 list of the worst faults found to date.
- Group faults into functional area and/or characteristics.

#### Where were faults introduced

- In which phase of the development did the faults originate and why.
- Which specific activity (e.g. interface design).

#### Identify the root cause of the most serious faults

- List all possible causes (e.g. design rules not followed).
- Identify the associated effects (failures).
- Identify cause and effect relationships.

Add fault categories and types to your fault catalog or taxonomy. A fault/defect taxonomy is used to get ideas about what to test, based on previously found problems.



TWEETABLE

**Chartered testing encourages testers to react on the response from the system and think for themselves.**

## 3.5 Release

When it is time to decide whether to release your product the following should be considered (also in conjunction with your previously defined test goals, the Definition of Done and the customer acceptance criteria):

#### Coverage

- > All planned test charters have been executed.
- > Sufficient test coverage has been obtained.

#### Quality

- > The probability of remaining faults has been reduced to a level that can be accepted by the customer.
- > The number of executed failure free tests indicates that a sufficient reliability has been reached.
- > All risks, hazards and safety issues have been evaluated to an acceptable/agreed level of confidence.

### Time

- > The deadline/ship date has been reached.

### Cost

- > The expected cost of correcting possible remaining faults is significantly lower than the expected return on sales.
- > The expected life cycle cost is within the limits of what your company and the support organization can handle.
- > The expected cost for the customer using your product including loss of data, efficiency, business and bad will is within acceptable limits.

To be “done” with a release means (the following release criteria example is based on [Rothman -01]):

- > All planned and agreed features includes fulfilling the required characteristics of the system.
- > All code must compile and build for all platforms.
- > No high priority bugs remain to be corrected.
- > For all open bugs, documentation in the release notes is included with workarounds (for the most important/serious bugs, if possible).
- > All planned tests run and passed.
- > The number of open defects has been decreasing for last three weeks.
- > Feature x unit tested by developers, system tested by testers, verified with customers A, B before release.
- > All open defects evaluated across all involved teams.

- > All User Acceptance Tests performed and passed.

- > All release documentation, user manuals and other system documentation reviewed, tested and ready for release.

When handing over the released system to your maintenance organization (if you have a separate one) make sure you save everything you have done in the project. That is, all test documentation, test data, test scripts, log files, test log notes, test reports, test environment and test tools. Make sure you have everything under configuration control so it is possible to trace which version of the software was tested against which version of the test data, test charters, test environment and its configuration.

After the release you should finish with a project retrospective to look back on what you learned during the project. The following description of a project retrospective is based on [Stevens -08].

Look at:

- > What happened?
- > What did we do well?
- > What could be improved?
- > Who has jurisdiction?
- > What is top priority?

A retrospective or lessons learned is normally performed at the end of each Sprint and at the end of the project. The Scrum master is the moderator for the meeting.

The result of the retrospective is two prioritized lists of action items of how to improve the productivity in the project. The first list contains actions that the team can perform on its own authority. Items from



the second list require agreement from somebody 'outside', e.g. Management, the Customer, another department, etc.

By asking what happened, you build a common understanding of the project. Just by getting people together, giving them a chance to talk, and have them listen to each other helps to reduce conflicts.

By asking what the team is doing well you may encourage and make sure good practices are kept by recognizing them.

By asking the team what and how to improve, you get a list of short and medium term action items which will make visible improvements in the state of the project.

For the suggested actions, make sure they are well described with sufficient detail to be able to determine if they were achieved or not when doing the next retrospective.

You may use the SMART concept (Specific-what, why, how, Measurable, Attainable, Realistic, Timely - a clear target) or just a clear direction of how to change to improve.

By determining who has jurisdiction, you can identify which items the team can do itself and for which items help or negotiations are needed from the outside. For the ideas the team are able to implement themselves, start with the improvements that give the most effect and are easiest to implement within the shortest time frame. For the rest, start with the top priority items and try to get them approved.

The team may have enough ideas to keep busy for a couple of months, so you will probably need to focus on just a few items so that the team can continue to do productive work. Less important items can be postponed to the next retrospective.

Also look at how efficient your testing were:

**1) Test coverage.**

What level of test coverage was possible to obtain for the tested User Stories, Features and Characteristics? What level would have been needed considering size, complexity, and risk?

**2) Test efficiency.**

How efficient was the testing in finding defects and execute tests? How much time was spent on debugging/adapting the test environment, test scripts or tools used?

**3) Confidence level in the test results.**

What was the confidence in the test results? Can they be trusted given the way the tests were performed? Is it easy to go back and check log files? Are there notes and time stamps to make backtracking possible?

**4) Failure rate, predictions and actual results.**

What was the actual failure rate, i.e. number of defects found per hour of testing?

**5) Defect detection percentage.**

What was the defect detection percentage at different types of testing (requirements review, unit, function and system testing)?

**6) Reasoning structure.**

How did the questioning and test ideas work? Were they efficient and did they reveal many problems?

**7) Test goals and strategies.**

To what extent were the test goals achieved? Were the test strategies defined in the initiation of the project used and efficient? Were many additional test strategies added along the way?

**8) Test plan fulfillment.**

Was the test plan for the project and for each sprint possible to follow? How much did it change? How much was written down? Can it be reused for the future?



TWEETABLE

During the end game the final regression tests and the finishing parts of the characteristics tests are performed.

**9) Planning precision (hit rate).**

What was the planning precision in each sprint, considering planned versus delivered User Stories, tested and passed demonstration and acceptance with the stakeholders?



TWEETABLE

The need for test coordination increases with the size of the project and the level of short term planning.

**10) Documentation and reporting.**

How did the test reporting work out? Did it create enough clarity for decision making regarding the quality of what was tested?



TWEETABLE

A Test Charter consists of test objectives, and test ideas used for exploratory testing.

**11) Test stop/release criteria analysis.**

Were the testing parts included and possible to determine in the analysis of the release criteria?



TWEETABLE

Use a “scientific” approach to explore the different parts in the system.

**12) Way of working.**

What way of working regarding testing was successful and not? What can be improved?

**13) Planned work items/tasks versus testing tasks actually needed.**

To what extent did the predicted work remaining for each testing task match reality?

**14) Velocity.**

Did the predicted velocity match the actual from the first release planning created during the project release planning in the beginning of the project? How much did it deviate from the actual? Did the actual velocity match what the project required from a testing point of view?

## 4. Biography



Since 1982 Anders has been working with testing of software and systems, mainly real-time and telecommunication systems, but also large client server systems. He is currently working as a consultant at Enea AB. where he works on test process improvements, test management, mentoring and test process implementation for different customers. Anders is also a teacher in test process courses - the ISTQB foundation certificate in Software Testing and Agile Testing. He presented this paper at EuroSTAR 2010 in Copenhagen.

## 5. References

- [Al-Qutaish -10] Quality Models in Software Engineering  
[http://www.jofamericanscience.org/journals/amsci/am0603/22\\_2208\\_Qutaish\\_am0603\\_166\\_175.pdf](http://www.jofamericanscience.org/journals/amsci/am0603/22_2208_Qutaish_am0603_166_175.pdf)
- [Bach1 -99] Risk and Requirements-Based Testing  
<http://www.cdainfo.com/Down/5-Test/Requirements.pdf>
- [Bach2 -01] What is Exploratory Testing?  
[http://www.satisfice.com/articles/what\\_is\\_et.shtml](http://www.satisfice.com/articles/what_is_et.shtml)
- [Bach3 -03] Exploratory Testing Explained  
<http://www.satisfice.com/articles/et-article.pdf>
- [Bach4 -10] Rapid Software Testing Appendices  
<http://www.satisfice.com/rst-appendices.pdf>
- [Bach5 -99] A Low-Tech Testing Dashboard  
<http://www.satisfice.com/presentations/dashboard.pdf>
- [Bach6 -10] Rapid Software Testing - Course Material  
<http://www.satisfice.com/rst.pdf>
- [Beizer -90] Software Testing Techniques, second edition ISBN 0-442-20672-0
- [Cagan -06] Assessing Product Opportunities  
<http://www.svproduct.com/assessing-product-opportunities/>
- [Claesson1 -02] A risk based testing process (QWE -02)
- [Claesson2 -02] How to use scientific methods in software testing (QWE -02)
- [Claesson3 -04] How to define efficient test goals and strategies (Eurostar -04)
- [Claesson4 -11] Course - Agile testing: Boost your testing efficiency  
[http://www.enea.com/templates/CourseDescriptionPage\\_\\_\\_\\_37960.aspx](http://www.enea.com/templates/CourseDescriptionPage____37960.aspx)
- [Cohn -10] How do story points relate to hours?  
<http://blog.mountangoatsoftware.com/how-do-story-points-relate-to-hours>
- [Cohn -05] Agile Estimating and Planning - Book  
ISBN-13: 978-0131479418, <http://www.planningpoker.com/detail.html>
- [Crispin /Gregory- 09] Agile Testing: A Practical Guide for Testers and Agile Teams  
<http://www.amazon.com/Agile-Testing-Practical-Guide-Testers/dp/0321534468>  
<http://lisacrispin.com/wordpress/presentations/>
- [DeBono -99] Lateral Thinking  
<http://www.solutioneers.net/solutioneering/lateralthinking.html>
- [Delphi -10] The Delphi project estimation/forecasting method  
[http://en.wikipedia.org/wiki/Delphi\\_method](http://en.wikipedia.org/wiki/Delphi_method)
- [Dubakov -10] How to Really Prevent and Manage Bugs in Agile Projects  
<http://targetprocess.com/LearnAgile/Whitepapers/AgileTesting.aspx>
- [Edgren -10] Test sampling and ground theory  
<http://thetesteye.com/blog/2010/05/sampling-serendipity/>
- [Ganesh -10] Evaluation of Software Quality using Quality Models  
[http://www.joyofprogramming.com/Doc\\_Presentations/QualityModel.pdf](http://www.joyofprogramming.com/Doc_Presentations/QualityModel.pdf)
- [GQM -10] Goal, Question, Metrics model  
<http://en.wikipedia.org/wiki/GQM>
- [Gregory -10] Agile testing  
<http://agile2010.agilealliance.org/files/Dance%20of%20the%20Tester%20-%20Janet%20Gregory.pdf>

- [Grindal -07] Handling Combinatorial Explosion in Software Testing  
<http://www.avhandlingar.se/avhandling/52f9532a06/>
- [Hazrati -10] Cost of Cross Functional Teams  
<http://www.infoq.com/news/2010/07/cost-cross-functional-teams>
- [ISO/IEC 9126-1] Software Engineering - Product quality - Part 1: Quality model  
<http://www.cse.dcu.ie/essiscope/sm2/9126ref.html>
- [Jeffries -01] Card, Conversation and Confirmation of User Stories  
<http://xprogramming.com/articles/expcardconversationconfirmation/>
- [Jo.Bach - 00] Session-Based Test Management  
<http://www.satisfice.com/articles/sbtm.pdf>
- [Kaner1 -04] The Nature of Exploratory Testing  
<http://www.testingeducation.org/a/nature.pdf>
- [Kaner2 -04] Software Testing as a Social Science  
<http://www.testingeducation.org/a/ifipkaner.pdf>
- [Kelly -08] Writing Good User Stories  
<http://www.agile-software-development.com/2008/04/writing-good-user-stories.html>
- [Kniberg -07] Agile Development  
[http://www.sast.se/q-moten/2007/stockholm/q2/2007\\_q2\\_kniberg.pdf](http://www.sast.se/q-moten/2007/stockholm/q2/2007_q2_kniberg.pdf)
- [Lambert -10] There's method in the madness  
<http://thesocialtester.posterous.com/theres-method-in-the-madness>
- [Marekj -08] Tester's view on XP's Card, Conversation, Confirmation  
<http://www.marekj.com/wp/category/watir/>
- [Poker -10] Planning Poker in Agile development projects  
<http://www.planningpoker.com/>
- [Rothman -01] Release Criteria: Defining the Rules of the Product Release Game  
<http://www.jrothman.com/Papers/ReleaseCriteria.pdf>
- [Schaefer -05] Risk Based Testing, Strategies for Prioritizing Tests against Deadlines  
<http://www.methodsandtools.com/archive/archive.php?id=31>
- [Stevens -08] Project retrospectives  
<http://agilesoftwaredevelopment.com/blog/peterstev/start-trust-start-retrospective>
- [Testeye -10] Software Quality Characteristics 1.0  
[http://thetesteye.com/posters/TheTestEye\\_SoftwareQualityCharacteristics.pdf](http://thetesteye.com/posters/TheTestEye_SoftwareQualityCharacteristics.pdf)
- [Wikipedia1 -11] List of system quality attributes  
[http://en.wikipedia.org/wiki/List\\_of\\_system\\_quality\\_attributes](http://en.wikipedia.org/wiki/List_of_system_quality_attributes)  
[http://en.wikipedia.org/wiki/Non-functional\\_requirements](http://en.wikipedia.org/wiki/Non-functional_requirements)
- [Wikipedia2 -11] Lateral Thinking  
[http://en.wikipedia.org/wiki/Lateral\\_thinking](http://en.wikipedia.org/wiki/Lateral_thinking)
- [Wikipedia3 -11] PDCA, The Deming cycle  
<http://en.wikipedia.org/wiki/PDCA>
- [Wiegers -03] Software Requirements – Second Edition ISBN 0-7356-1879  
<http://www.processimpact.com/pubs.shtml>
- [Wiegers -10] Requirements process guidelines and templates  
<http://www.processimpact.com/goodies.shtml>

# Join the conversation...

If you've enjoyed this eBook, then come and interact with the EuroSTAR Community! You'll find great Software Testing content, and great minds that are passionate about software testing.



Follow us on **Twitter** @esconfs

*Remember to use our hash tag #esconfs when tweeting about EuroSTAR 2011!*



Become a fan of EuroSTAR on **Facebook**



Join our **LinkedIn Group**



Keep up to date with the **EuroSTAR Blog**



Subscribe to our Newsletter - **STARtester**



Check out our free **Webinar Archive**

**[www.eurostarconferences.com](http://www.eurostarconferences.com)**