ESPA Guidebook: At the Source of Quality Practices

Timo Lehtinen (Ed), Jari Vanhanen, Juha Itkonen, Eero Laukkanen. ESPA Guidebook: At the Source of Quality Practices. 2012.

Table of Contents

1 Introduction

Lots of research has been conducted on various quality practices, i.e. practices that aim at ensuring the software quality. However, current knowledge of their true costs and benefits, and even what kinds of quality problems they help solve is scarce and shallow. Current research of quality practices is in many cases based on student experiments where individual quality practices are studied in artificial context for small tasks. Furthermore the results of such studies may act as poor substitutes of the practical results practitioners are interested in.

In this book, we discuss the costs and benefits of quality practices in several software companies. Previous industrial research is often based on data from individual projects and generalizing the often conflicting results is difficult due to poor understanding of the dissimilarities between the contexts. This longitudinal study lasting almost four years, made it possible to create close co-operation with the software companies allowing understanding the effects of the context on the results and generalizing the results.

Our work with Finnish software companies has revealed that companies typically do not consciously select and improve the quality practices they use to reach their quality goals, nor do they have the knowledge needed to do so. Thus, deeper empirical evidence of quality practices is needed to help companies select a set of quality practices for reaching their desired quality goals.

Existing empirical evidence of quality practices can help select a better initial set of practices, but any single set of “best practices” does not work for all cases. Continuous collection and analyzis of quality information, (e.g. the number and type of defects found using code reviews, test coverage of unit tests) would allow further improving the set of quality practices and their application. Ideally, quality information would allow software development organizations to find the most efficient quality practices and reach the desired quality goals with higher probability and lower costs. One

example of this is IBM[1], where the methodologies of root cause analysis have been applied to quality information on software defects resulting to lower defect rates and increased productivity caused by the software process improvements made. Unfortunately, quality information is insufficiently collected and utilized in many software development organizations due to difficulties in data collection as well as a limited understanding of the potential uses of the information. The need of a method for goal and evidence-based quality practice selection and improvement elevates together with the need of deeper empirical evidence of quality practices as well as guidelines for increasing the utilization of quality information and for simplifying its collection.

The main goal of this book is to present a practical method for goal and evidence-based quality practice selection and improvement. The main goal consists of the following sub goals (see Figure 1):

1. to help select quality goals and related quality practices for different contexts based on easily accessible empirical evidence of the quality practices
2. to increase generic empirical evidence of the benefits, costs and context dependencies of two quality practices: intelligent manual testing and pair programming
3. to help utilize quality information to further improve the set of selected quality practices and their application, as well as to daily software development

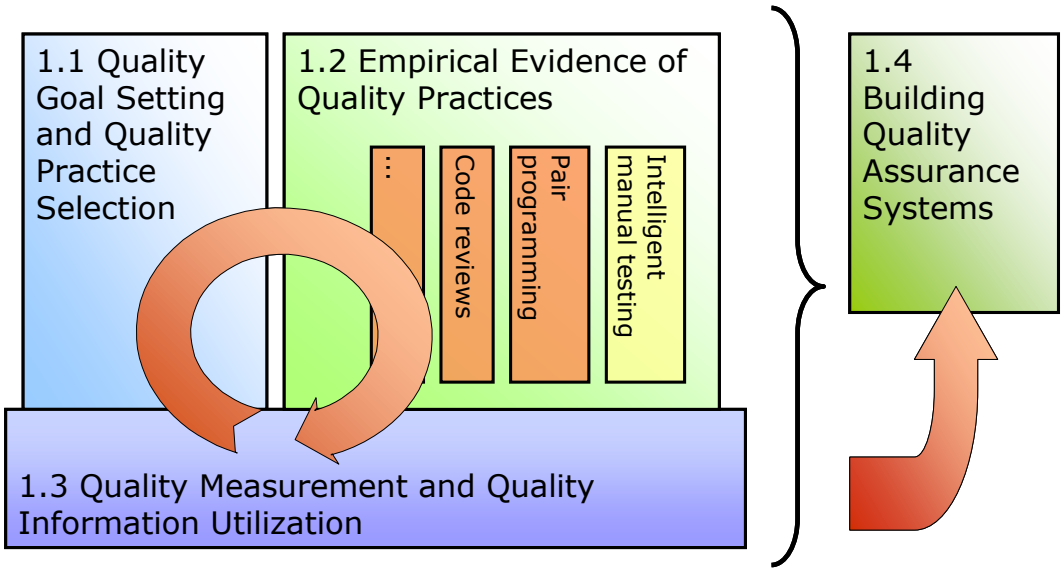

Figure 1 Framework for product quality

[1] Mays, R.G. 1990, "Applications of Defect Prevention in Software Development", *IEEE Journal on Selected Areas in Communications*, vol. 8, no. 2, pp. 164-168.

1.1 Quality Goal Setting and Quality Practice Selection

This book explains how to identify, prioritize and document quality goals for the purposes of selecting a sufficient set of quality practices for reaching the quality goals. Making informed decisions on how to ensure quality in a certain context requires identifying the quality goals. The quality goals together with the empirical evidence of various quality practices makes it possible to make an informed, initial selection of quality practices that can help in achieving the quality goals.

The identification of the quality goals means how the context specific quality goals can be discovered. Prioritization and documentation of quality goals refers to understanding what kind of prioritization and level of documentation supports the selection of quality practices. Finally, we introduce viewpoints for selecting quality practices based on prioritized and documented quality goals.

1.2 Empirical Evidence of Quality Practices

Many quality practices are used in industry and discussed in the literature. Empirical evidence of quality practices includes their benefits, costs, limitations and context dependencies. Empirical evidence is required to be able to select right practices for a certain context. However, existing empirical evidence is scarce and shallow and does not provide enough information for optimal selection of quality practices for a certain context.

Almost all software development relies, at least partly, on manual testing, and it is likely that it will do so for the foreseeable future. However, testing research has close to ignored the intricacies of manual testing, something we hope to partially rectify by focusing on “Intelligent manual testing” as one of the quality practics discussed in this book.

Testing can be seen as creative work, rather than as a mechanical and repetitive task. Intelligent manual testing (IMT) relies on human experience and skills that allow greater flexibility and effectiveness than traditional manual test-case-based testing. Exploratory testing, which is one form of IMT, has been introduced by testing practitioners and consultants who openly advocate its benefits, but confirmation of these benefits through objective studies are still missing. IMT focuses to the testers’ test execution practices, techniques and defect detection strategies – instead of pre-design, test cases and details of test documentation.

As another example of quality practices, this book discusses pair programming, which is a practice where two developers work together at the same workstation. The main reasons for using it are the expected increases in software quality and knowledge transfer within the development team.

1.3 Quality Measurement and Quality Information Utilization

Quality information means any information related to the quality of the product, e.g., number and type of defects found in a product after the release, or related to the use of quality practices, e.g., the effort spent for IMT sessions during a release. In our experience, quality information is typically insufficiently collected and utilized in software development organizations. It is used mostly for daily software development, e.g., managing testing tasks or managing defects, even though the information also could be utilized for selecting quality practices and improving their application.

An import aspect for quality measurement and quality information utilization is the reduction and simplification of the work required to collect relevant quality information that can be used for daily software development, for selecting quality practices, and for improving their application. The key is to understand how and what quality information can be utilized for different purposes. For example, what information helps to focus testing resources, or what kind of defect classifications provides relevant information. We emphasize the simplicity and validity of quality information, and low costs for data collection e.g. through the use of commonly existing data sources.

Finally, managing testing tasks and defects is important, but highly reactive. Utilizing quality information in more proactive ways helps to avoid the need of reactiveness during the development. Root cause analysis is a method for utilizing quality information resulting to better quality practice selection and process improvements. Learning from the prior projects results to success in the future projects. The weaknesses of current quality practices can be detected by focusing on the causes of quality deviations and utilizing quality information of people as a source of knowledge.

1.4 Content of this book

There are seven chapters in this book and they follow the general framework for software quality (see Figure 1). Chapter 2 introduces a lightweight method for setting up the quality goals of your software project.

Chapter 3 gives you an overview for software quality practices by connecting explicitly the quality practices and qoals. Thereafter, Chapter 4 connects the quality information to quality goals and practices by introducing a lighweight root cause analysis method for quality practice improvements and selection. Chapters 5 and 6 introduce two quality practices. Chapter 5 introduces intelligent manual testing and Chapter 6 introduces pair programming. Finally, as being highly important part of product quality information, Chapter 7 presents challenges and guidelines for defect reporting.

2 Defining Software Product Quality Goals

2.1 Introduction

The definition of software quality is always dependent on the context. Requirements stating desired values for quality attributes are called non-functional requirements, quality requirements or quality goals.

In the context of market driven software products deciding on the most relevant quality goals is more challenging than in customer specific software project business. In a customer specific project there is always a voice of the customer who can be used to decide in the tradeoffs of various quality goals. In a market driven context product management must decide, based on the needs of the various customers, what qualities to strive for. Furthermore, software product companies cannot neglect the internal quality attributes such as maintainability or testability because that might result in a rapid increase in software development costs.

After deciding on the quality goals they need to be achieved. There are at least three alternative or complementary approaches that contribute to achieving the quality goals. The first approach is to build a quality goal in the product as a functional requirement, e.g., by developing installation software for decreasing update effort. In the second approach quality goals are considered in the product architecture (Barbacci, et al. 2003), e.g., by designing the architecture to support faster manual updates. The third approach is to adopt software development practices that contribute to achieving the quality goals (Wagner et al. 2008, Järvinen at al. 2000). For example, in our previous work we found that code reviews improve maintainability (Mäntylä, Lassenius 2009), pair programming increases understanding of code and thus contributes to maintainability (Vanhanen, Lassenius 2005), and exploratory testing detects issues relevant to end-users such as usability problems (Itkonen, Rautiainen 2005, Itkonen et al. 2007). We use term quality practice to mean any practice that contributes to building or assuring software quality as its primary or secondary

purpose. We look at the elicitation and analysis of quality goals especially for the purposes of the third approach.

The elicitation of quality goals is a challenging task for which several methods have been proposed in the field of requirement engineering, e.g., (Cysneiros, do Prado Leite 2004, Herrmann, Paech 2008, Mylopoulos et al. 2001, Doerr et al. 2005). However, these methods seem to be very comprehensive, complex and thus quite possibly laborious to learn and use. If simple methods do not exist, companies may not use any method and quality goals are found too late or not at all as was noticed in two industrial case studies (Borg et al. 2003). Furthermore, for the purposes of process improvement it may not be necessary to define quality goals on the same level of detail as in a rigorous requirements specification. It can already be valuable to identify the most important subset of quality goals and even these only on a general level. For example, realizing that usability and security are the most important quality goals can steer the selection of software development practices without refining the quality goals any further.

Below we present a method that can be used to elicit, prioritize, and elaborate the quality goals of a software product. It is designed to be lightweight and easy to learn compared to methods for a more comprehensive analysis of non-functional requirements. The method and the resulting quality goals are meant especially for improving the selection and improvement of quality practices. We also present experiences of using it in four software product companies.

2.2 Description of the method

The method consists of a workshop where quality goals are elicited through brainstorming and prioritized by voting. Finally, the most important goals are elaborated. The method borrows the ideas of arranging a workshop involving all relevant stakeholders, and of using voting for prioritizing quality goals from the Quality attribute workshop (QAW). The quality indicator concept to be used in the elaboration of the goals is borrowed from the QUPER model (Regnell et al. 2007).

2.2.1 Definition of a quality goal

In our terminology, a quality goal is a description of the desired state of some internal or external quality characteristic of a software product including those listed in ISO 9126 quality model (ISO/IEC 9126-1, 2001).

By software product we mean the executable software, its source code, and documentation.

In order to keep the focus manageable, we explicitly exclude other types of goals that are often touched when discussing about quality. Goals related to the development process, such as organizing software testing are excluded, since they typically already characterize solutions for achieving some quality goals. Goals related to services such as customer support or user training, and general project goals such as schedule compliance are also excluded.

For example, customer satisfaction is not a quality goal, but it may depend on several factors, e.g., absence of critical defects, delivery project's schedule compliance, and quality of user training. Only the first of these is a quality goal.

2.2.2 Objectives for the method

We aimed at creating a lightweight and easy to learn method for eliciting, prioritizing, and elaborating quality goals for a new or existing software product either on the product level or for a specific product release project. The planned main purpose for the resulting quality goals is to facilitate the selection and improvement of quality practices. Other uses can be to increase knowledge of the quality goals by communicating them better and to improve observing the achievement of the quality goals.

The planned purpose for the quality goals means that it is more important to understand which quality goals are important, and what they roughly mean for the product than being able to state accurate target values for each quality goal. The practices needed are probably very similar as long as the target values are roughly accurate.

The method was created for organizations that may have quite limited amount of effort available for the process improvement activities. Therefore the application of the method should not require too much effort. The method should also be simple enough to learn and use so that it can be used in organizations without the help from external consultants.

2.2.3 Participants

The method consists of preparing and arranging a workshop for a group of people who represent all important viewpoints for eliciting and analyzing quality goals in the chosen context. At least one participant should be assigned to represent each of the following viewpoints: business, end user, project management, development and quality assurance. End users are

represented by someone from the company, such as the product manager, instead of real end users in order to keep the discussion honest and open. One person can take more than one viewpoint if that is necessary to cover all the viewpoints. Someone needs to take the role of the workshop chair.

2.2.4 **Steps of the method**

Step 1: Preparation

The workshop chair sends an invitation to the participants. It contains a short introduction of the workshop and a request to list briefly ideas of important quality goals from their viewpoint and to send them to the chair before the workshop. This is supposed to start the mental preparation and processing of the quality goals before the workshop and to shorten the duration of the workshop. If the workshop is arranged by outsiders, this preparation allows them to get some idea of the quality goals beforehand.

Step 2: Introduction

The chair begins the workshop by introducing its purpose and content. The term quality goal is introduced through exemplary quality goals that emphasize the characterization of the product, and through counter examples of goals that are not quality goals. The clear definition is given in order to avoid listing general project goals, goals for development processes, or goals related to any services such as training.

The context for which the quality goals are to be defined is discussed briefly in order to ensure that all participants have the same conception of it. For example, what is the product for which the goals are to be set, and is the purpose to think about the goals for the next release or in a longer term?

Step 3: Elicitation

The quality goals are elicited using a simple brainstorming technique. The chair asks the participants to think what the important quality goals of the product are, and write down each idea on a sticky note. The ideas listed during the preparation phase should also be included. People have a tendency to focus on goals that require improvement. Therefore, if you want to elicit also those quality goals that are already on adequate level it is better to ask participants to first list such goals. Only after that ask for those quality goals that need improvement.

When the participants seem to be ready, each sticky note is put on the wall with a short verbal description by its author. The participants may continue writing sticky notes, if they get new ideas.

If someone proposes an idea that is not related to a quality goal, but describes, e.g., a general project goal, the chair asks the author to tell how it depends on some quality goal(s). If a dependency is found, the idea is modified to refer directly to the related quality goal.

Continuously, the chair groups and labels sticky notes that are related to the same topic, e.g., usability or performance, and removes duplicates. Grouping decreases the amount of quality goals to a manageable level, and should make the groups about similar in their abstraction level so that there is sense in prioritizing between them. Each group becomes a separate quality goal that is described by the grouped sticky notes. The sticky notes are used later when elaborating the quality goals.

When all the ideas are on the wall, the chair uses the ISO 9126 list of quality characteristics and their sub-characteristics (ISO/IEC 9126-1, 2001) as a checklist. He matches the quality goals under the most appropriate ISO 9126 quality characteristics and asks the participants to ensure that they have not forgotten any relevant quality characteristics. The quality model is only shown afterwards in order to avoid restricting ideas to any predefined categories and to make the participants contribute personally rather than select items from a predefined list. When people have personally contributed they are less likely to suffer from the not invented here syndrome.

Step 4: Prioritization

The quality goals are prioritized through voting from the viewpoint of their importance to the product. All participants have the same number of votes, e.g., as many as there are quality goals. A participant can cast zero or more votes to each quality goal. Voting is done publicly by drawing lines on the wall next to the quality goals. The participants are asked to consider their assigned viewpoint when voting in order to ensure that all viewpoints are covered. The total number of votes given per quality goal defines the priority order.

Step 5: Elaboration

Only the most important quality goals are elaborated in order to keep the duration of the workshop moderate. However, anyone can propose including some lower priority quality goals in elaboration, if he/she can give a good reasoning for doing so.

The chair introduces the quality goal elaboration template and the ideas of the QUPER model (Regnell et al. 2007). The quality goal attributes to be analyzed and documented are listed in Table 1. They include, e.g., the description and rationale of the goal, and quality indicators (QI) with most of their attributes from the QUPER model.

Table 1 The attributes of a quality goal

Attribute	Description
Name	Short name.
Description	Meaning of the goal.
Rationale	Gains and risks. Why should we aim for this goal? What if we don't reach it?
Related factors	Tentative ideas that prevent/contribute reaching this goal.
Votes	Number of votes given to the goal.
Quality indicators (QI)	One or more quality indicators per goal.
QI: description	Brief description.
QI: breakpoints	Estimated values of utility, differentiation and saturation breakpoints.
QI: cost barriers	Factors causing a steep rise in cost when targeting beyond a certain quality level.
QI: current level	Current quality level.
QI: target level	Target quality level at a certain time.

The quality indicators should be concrete enough to be measurable at least in theory. However, in our method their main value is to communicate the most important aspects of a quality goal. Being able to measure them objectively and accurately is not as crucial.

The participants are divided into groups of at most three persons. The groups share the quality goals so that each group contains suitable roles for analyzing the assigned goals. Each group discusses for about thirty minutes per quality goal and documents their attributes to a spreadsheet template. Finally each group briefly presents their results to the other groups. After the workshop, the chair and possibly some other participants finalize the documentation of the quality goals, and communicate them to the whole organization.

2.3 Examples of using the method in four different cases

2.3.1 Overview of the cases

We applied the method in four cases in four different companies (see Table 2). In each case the context was related to a mature product that was one of the main products of the company. In cases 3 and 4 the context was the product in general. In cases 1 and 2 it was a specific project related to the product. In case 3 the product is installed by the customer, but in the other cases a delivery requires configuration, integration and installation work from the supplier even though customer specific feature development

is mainly absent. The customer base of each product ranges from a few dozens to a few hundreds.

Table 2 Summary of cases

	Case 1	**Case 2**	**Case 3**	**Case 4**
Context	main product's delivery projects to a certain group of countries	a release project combining functionality of another vendor's product to own main product	one of company's main products in general	one of company's main products in general
Participants	7	3–5	4–7	5
Ideas	~50	~50	~50	~40
Quality goals	14	8	14	13

In each case 2–3 researchers participated in the workshop and one of them acted as the workshop chair. The workshop duration was fixed to about 4 hours, which meant 20–30 man hours of effort for the company personnel. Keeping the schedule was ensured by adjusting the number of elaborated quality goals.

In all cases the participants were able to represent all the desired viewpoints. Their roles included e.g., CEO, product development manager, project manager, quality assurance manager, developer, tester, UI specialist, and technical customer consultant.

In each case brainstorming produced 40–50 ideas of which a few were clearly unrelated to quality goals. We did not remove them, but when the ideas were grouped and matched to ISO 9126 categories, they were disregarded as they did not match any category.

Table 3 An example of a quality goal

Attribute	**Description**
Name	Updating
Description	Updating the software should be quick and easy, ideally possible without deep technical or product knowledge.
Rationale	Direct cost savings related to updates. More personnel capable of updating. Reduced risk of errors during updates.
Related factors	Software robustness, configurability, use of installer software.
Votes	6
QI1: description	Work time consumed by update.
QI1: breakpoints	3 hours, 1 hour, 10 minutes
QI1: cost barriers	After useful: work related to defining default configuration, after competitive: setting up automatic installers.
QI1: current level	2 hours
QI1: target level	30 minutes
QI2: description	Number of manual steps in update.
…	…
QI3: description	Number of problems found after update caused by errors done during update
…	…

In all the cases, the output of the method was a spreadsheet that contained 8–14 quality goals of which 2–5 were elaborated. Even though the products in the cases were quite different from each other, many quality goals were common to all cases. Usability, installability/updateability, functional correctness, functional suitability and performance efficiency were all among the ten most important goals in at least three cases. Table 3 shows an example of an elaborated goal.

2.3.2 **Case 1**

In case 1, we had seven participants all representing at least somewhat different roles. Most of them represented management, but the representatives of quality assurance and software development had firsthand knowledge also of the practical work.

We noticed that the participants seemed to focus on quality goals that needed improvement and to neglect quality goals that were important for the product, but were already on adequate level. Therefore in the end of brainstorming we explicitly asked the participants to list the current success factors related to product quality. Then the product development manager added two important goals. However, they did not receive many votes from the other participants. Because the success factors were neglected we decided to emphasize in the next cases that all important quality goals regardless of their current state should be elicited.

During the elaboration some goals turned into process goals instead of quality goals. It seems that people have a tendency towards thinking about solutions before the quality goals are analyzed. For example, testing became a quality goal having indicators defining targets for the amount of tests executed, for the efficiency of testing as a ratio of internally found bugs of all reported bugs, and for the duration and effort of a testing cycle. Based on the documented rationale, the real quality goal seemed to be partially testability and partially functional correctness. Testing on the other hand is one of the quality practices which aim at improving functional correctness.

Based on this experience we changed the method for the next cases so that the chair names the groups of sticky notes immediately when forming them, instead of letting the other participants do it when elaborating the goal. This ensures that the names reflect some quality goal, and elaboration is done from the correct point of view. We also decided to take a more active role during elaboration in order to ensure that elaboration focused on those issues we thought were relevant.

The company utilized the output to decide on a set of actions to help them achieve some of their most important quality goals. The identification of

quality goals and actions clarified and concretized their ideas of quality improvement. It put the quality issues on the open instead of leaving everyone nagging in they own offices. The process of eliciting and analyzing quality goals widened their view of quality from just looking at each quality problem separately.

2.3.3 **Case 2**

In case 2, there were first only three participants, all managers, who had a good insight into the practical development and quality assurance work. After the prioritization step a developer and a UI specialist joined the workshop. The reason for their shorter participation was the company's desire to save resources spent for the workshop.

It still seemed that listing important goals that were already on an adequate level did not happen in balance with those goals that needed improvement. There were first only a few of them, and some were added when asked explicitly after the brainstorming.

Developers who joined the workshop after voting, seemed to accept the relevancy of the quality goals and their priority order, or at least they did not question them when they were asked to comment and add goals.

During the elaboration we found that one of the goals to be elaborated ("usability") included too many different ideas. It was split into four separate goals of which only the most important one was elaborated.

2.3.4 **Case 3**

In case 3, the workshop split into two parts due to a shortened time slot for the workshop. The elaboration step was done a week later. This unplanned break was actually considered a good thing by the participants for recharging their brains for active participation.

There were seven participants in the elicitation step and four of them were able to participate in elaboration. In both sessions all relevant roles were still represented.

We made one major change to the method after the previous case. Brainstorming was started by explicitly asking for the important quality goals that were already on adequate level. Only after that we asked for those quality goals that need improvement. This order seemed to generate much more ideas that described quality goals that were already on adequate level. At least one third of the ideas were now such ideas.

In order to ensure that the results were considered in the development projects, the company organized a further workshop in the context of a

specific release project. For each feature area, 1–4 most relevant product level quality goals were selected and concretized for the release project. The participants ended up documenting only the name and concretized description of the quality goals. Explicit quality indicators were not conceived, because the quality goals were already quite concrete. Some of the goal descriptions characterized target values, though not measurable ones. This kind of elaboration seemed to work well in connecting the product level quality goals to the actual product features. In addition, the quality practices for achieving the quality goals were identified. This led to tasks in the project plan, which concretized how the goals were to be achieved in terms of development and quality assurance tasks.

2.3.5 Case 4

In case 4, we ran the workshop again in one session, and there were five participants representing all relevant stakeholders. The method was used in the final form as described in section 2. We were satisfied with the output and how the workshop went.

2.4 Lessons learned

Below we summarize lessons learned from the four cases. The main lessons are summarized in Table 4.

We noticed that when the context was a specific release project (case 1 and 2) instead of a product in general (case 3 and 4), the discussion of quality goals still often drifted to the product in general. Therefore, we recommend that the product management process should first elicit and analyze the quality goals for a product in general and then arrange another workshop(s) for specific release projects. The product level quality goals can act as the basis for eliciting and/or analyzing quality goals for the more specific context.

In addition to the quality goals that need improvement, identifying the current success factors was considered important by the companies, or otherwise preserving them may be forgotten. For example, if a company's key success factor has been usability, it must not be forgotten in the future as competitors are continuously trying to close the gap and as it may decay for the new features unless it is actively paid attention to. However, in cases 1 and 2 the participants seemed to focus on those quality goals that need improvement and to neglect those that already were success factors of the product. Asking the success factors explicitly before starting to list improvement targets worked as a solution to this problem in cases 3 and 4.

Despite of our attempts to emphasize that quality goals should be related to the product, some process related ideas were still proposed in all cases. These were related to, e.g., schedule and process conformance. We did not remove them during the workshop, because we did not want to discourage the authors by saying that their goals are not correct quality goals. In two cases a process goal even became a high priority goal, but in case 1 “testing” goal was later modified to “testability” quality goal and in case 4 “system test organization” goal was removed among the most important goals after the workshop when finalizing the documentation of the quality goals.

We presented the ISO 9126 quality model to the participants only after the free brainstorming of the quality goals in order to avoid limiting the creativity of the participants. In each case it generated only a few or no new quality goals. It seems that a group of people representing all relevant viewpoints can quite well cover important quality goals even without a checklist. On the other hand most of the elicited goals, except the process related goals that we actually did not even want to include, seemed to relate to some quality characteristic in ISO 9126 if interpreted loosely enough. Therefore it seems unlikely that we had missed some goals, even if the brainstorming had been done by going through the quality characteristics of the ISO 9126 quality model.

There are always challenges in the prioritization of goals related to considering different perspectives. These include, e.g., the importance of the quality goals regardless of their current state, the importance of improving the quality goals, and the capabilities of different people to see the overall importance of all quality goals. Some goals may be inter-related, e.g., a quality goal may be important from the business point of view, but it may depend on another quality goal that can be identified only by the developers.

When elaborating the quality goals, the identification of the quality indicators seemed to be helpful. Bare name and description of a quality indicator already clarified the otherwise vague quality goals as can be seen in the example in Table 3.

It seemed to be difficult to estimate the current quality level for some quality indicators. Often analyzing data from some information system would have been needed. Sometimes it was acknowledged that there was no data available anywhere and participants have difficulties in estimating the current value. If quality improvement should be based on analyzing how the implemented changes to the process affect the quality, improvements to the measurement of the quality indicators would be needed in these cases.

The participants should have enough authority to represent their role and thereby their group can accept the results of the workshop. Thus, it is

not enough that only a random representative of, e.g., software testers participates. The representatives must have the respect of their peers or otherwise the results of the workshop might be rejected.

The realized effort for eliciting and analyzing the quality goals was considered quite large (20-40 man-hours), but the results were also considered good. The required effort was hoped and expected to be lower, somewhere around 5–10 man-hours, when utilizing the product level quality goals for eliciting and analyzing the quality goals for a new release project, and also when revising the product level goals in the future.

Table 4 Tips for applying the method

- **Product level first.** It seems that it is better to set quality goals first on a product level before going to the project level. Otherwise, the discussion easily drifts from a specific project to the product level anyway.
- **Current success factors are easily ignored.** Identifying already achieved quality goals, in addition to those that needed improvement, was considered relevant, but was ignored unless explicitly considered.
- **A quality goal checklist is not necessarily needed.** Using ISO 9126 quality model as a checklist after the free brainstorming of quality goals produced only a few, if any, new, relevant quality goals. However, using some checklist as a basis for brainstorming might be an improvement to the faced challenges related to the lack of common terminology for discussing quality goals and for filtering away goals that are not related to the product.
- **Clear perspective for prioritization.** There are numerous relevant perspectives for prioritizing. For example, short term customer needs vs. long term maintainability goals, or absolute importance of a goal neglecting the current state vs. considering the size of the gap between the current and desired state for a goal. The perspective(s) to be used certainly depend on the situation but in any case, the desired perspective for prioritization needs to be communicated clearly.
- **Measures increase understanding.** Some elaboration of the quality goals, especially the general descriptions of the quality indicators seemed to improve the understanding of the quality goals. On the other hand, some of the quality goal attributes were not useful for facilitating the selection of new quality practices, which indicates that they were too detailed for that purpose.

References

Barbacci, M.R., Ellison, R., Lattanze, A.J., Stafford, J.A., Weinstock, C. B., and Wood, W.G., Quality Attribute Workshops (QAWs), CMU/SEI Technical Report, 2003.

Borg, A., Yong, A., Carlshamre, P., Sandahl, K., "The Bad Conscience of Requirements Engineering: An Investigation in Real-world Treatment of Non-functional Requirements" Third Conference on Software Engineering Practice in Sweden, 2003, pp. 1–8.

Cysneiros, L.M., and do Prado Leite, J.C.S., "Nonfunctional Requirements: From Elicitation to Conceptual Model", IEEE Transactions on Software Engineering, 30(5), 2004, pp. 328–350.

Doerr, J., Kerkow, D., Koenig, T., Olsson, T., Suzuki, T., "Non-functional requirements in industry - three case studies adopting an experience-based NFR method", IEEE International Conference on Requirements Engineering, 2005, pp. 373–382.

Herrmann, A., and Paech, B., "MOQARE: misuse-oriented quality requirements engineering", Requirements Engineering, 13(1), 2008, pp. 73–86.

Itkonen, J., and Rautiainen, K., "Exploratory Testing: A Multiple Case Study", International Symposium on Empirical Software Engineering, 2005, pp. 84–93.

Itkonen, J., Mäntylä, M.V., and Lassenius, C., "Defect Detection Efficiency: Test Case Based vs. Exploratory Testing", International Symposium on Empirical Software Engineering and Measurement, 2007, pp. 61–70.

ISO/IEC, ISO/IEC 9126-1:2001, Software Engineering–Product Quality–Part 1: Quality Model, Int'l Organization for Standardization, 2001.

Järvinen, J., Komi-Sirviö, S., Ruhe, G., "The PROFES Improvement Methodology - Enabling Technologies and Methodology Design", International Conference on Product Focused Software Process Improvement, 2000, pp. 257–270.

Mäntylä, M.V., and Lassenius, C., "What Types of Defects Are Really Discovered in Code Reviews**?**" IEEE Transactions on Software Engineering, 35(3), 2009, pp. 430–448.

Mylopoulos, J., Chung, L., Liao, S., and Yu, E., "Exploring alternatives during requirements analysis", IEEE Software, 18(1), 2001, pp. 92–96.

Regnell, B., Höst, M., and Berntsson Svensson, R., "A Quality Performance Model for Cost-Benefit Analysis of Non-functional Requirements Applied to the Mobile Handset Domain", 13th International Working conference on Requirements Engineering: Foundation for Software Quality, 2007, pp. 277–291.

Vanhanen, J., and Lassenius, C., "Effects of Pair Programming at the Development Team Level: An Experiment". International Symposium on Empirical Software Engineering, 2005, pp. 336–345.

Wagner, S., Deissenboeck, F., and Winter, S., "Managing quality requirements using activity-based quality models", 6th International Workshop on Software Quality, 2008, pp. 29–34.

3 Introduction to Quality Practices

This chapter focuses on quality practices in iterative and incremental software development. The viewpoint in this chapter is agile. First we must cover some fundamental concepts to clarify how the writers of this book see software quality, quality assurance and quality practices.

Software quality is a multifaceted concept that can mean very different things to different people. Software quality is impossible to define in general way, because the application domain, used technology, environment, users and customers, and many other things affect the definition of good enough or acceptable quality of a software system in each case and each context. Consequently, we either won't provide a universal definition of software quality. Instead, we encourage software developers and product managers to define the actual quality properties that are important for their specific software products and projects. Every software organization should think what high quality means for their software and their customers, instead of crying for a universal definition or international standard for high quality software. The specific definition of high quality in the software development organization also helps to answer the question: "How can we measure the quality of our software?"

Software quality standards, such as ISO 9126, provide good general classifications of the relevant quality attributes and metrics for software. These standards, however, try to capture the whole spectrum of different possible characteristics of software that can be considered to stand for high quality. This means that the standards cannot help in understanding the most important quality characteristics of a certain product or software system in a specific context.

Software quality assurance can be seen as an umbrella activity enclosing all processes and practices that are needed to ensure that the developed software fulfils the set quality requirements. In this book we do not cover software quality assurance (SQA) in the traditional sense. Traditional SQA approach is based on defined processes and standards and the goal of SQA is to ensure that the processes and standards are followed, measure deliverables, and react to discrepancies. In this book we concentrate on what we call *quality practices*.

Quality practices are the actual means that are used in software development work to achieve good enough quality level and evaluate the achieved quality level. Quality practices are activities performed by developers, testers, designers, analysts, project managers, etc., i.e. the same people who do the actual software development work. The activities of a separate SQA organization, that does not participate the software development work by themselves, are not the topic of this book.

3.1 Time-Paced Quality Practices Framework (Second Edition)

This section describes a Time-paced Quality Practices Framework (TQP). TQP is a general framework for describing and understanding quality practices as part of time-paced iterative and incremental software development. The TQP framework is based on the CoC framework, which is a general framework for describing iterative and incremental time-paced software development based on the concept of time pacing. Time-pacing refers to the idea of dividing a fixed time period allotted to the achievement of a goal into fixed-length iterations. At the end of each iteration there is a control point, at which progress is evaluated and possible adjustments to the plans are made. Changes can only be made at such a control point. This both accomplishes persistence and at the same time establishes flexibility to change plans and adapt to changes in the environment at the specific time intervals. (See Part I for more detailed description of IID and time-pacing in general.)

TQP is a framework that can be used to understand quality practices as part of time paced IID process. The framework consists of three main parts: *Quality Palette*, *Rhythm*, and *Viewpoints*. Quality palette is used to describe and prioritize the quality goals and connect quality practices for achieving the goals. Rhythm is used to connect the quality practices to iterative and incremental software development process and understand when the practices are applied. The viewpoints are applied to evaluate the current practices from different angles and find weak spots, risks and improvement opportunities.

3.1.1 Quality Palette

Summary of the organization's quality practices can be presented in convenient form using *Quality Palette Matrix*. Quality palette connects explicitly the quality practices and goals. Quality Palette can be used to assess the usefulness or appropriateness of the quality practices in terms of the stated quality goals. It also works as a tracking sheet for evaluating the

actual usage of the selected practices. An example of a quality palette is presented in Figure 2.

Practice	Amount of use	Quality Goals			
		Easy usability for occasional user	Correctness of calculations	Performance	Maintainability and extendability
Usability evaluation	1	++			
Alpha and beta test programs	2	+	+		
Functional system testing	2	+	+		
Automated tests	0		++		
Performance metrics	1			++	
Scenario tests	2			+	
Code reviews	1		++		+

Legend
++ = Good practice
+ = Helps
2 = Requlary used
1 = Occasionally used
0 = Not used

Figure 2 Example of a Quality Palette Matrix

Quality Palette facilitates discussing and defining quality goals for the software system or product in question. One way of creating the goals is to arrange a workshop where different stakeholders of the software project participate in defining and prioritizing the quality goals. A common challenge in defining quality goals is how to leave anything out (i.e. all quality characteristics are important). In practice, however, we cannot focus to all possible quality characteristics at the same time. By selecting the most important quality characteristics as our goals we can focus to those and get improvements done in the most critical areas first.

In the quality palette matrix the applicability of the quality practices for each stated quality goal are analyzed and marked to be a good practice for achieving the goal in question, a practice that helps to achieve the practice in question, or no relationship at all.

In addition, the matrix can be continuously used to track the actual usage of the practices. This can be done, for example, by assessing the usage in a reflection meeting arranged after each iteration. In the same meeting also the applicability of the practices can be re-evaluated based on the fresh experiences.

Create a quality palette matrix for your own project

- Estimate how well the identified practices work?
- Evaluate how rogorously the practices are applied?
- How well the practices support the most important quality goals?
- Are there weakly supported goals? What new practices could be taken into use?
- Are there very central key practices that contribute to many quality goals?
- Are there unimportant practices? What practices could be dropped?
- Can the practices be adjusted or tailored to better match your neeeds?

When analyzing the Quality Palette we can identify different anomalies that might be good candidates for improvements. Examples of such anomalies are:

- Quality goals that have no good practices
- Practices that do not contribute to any quality goal
- Practices that contribute to many quality goals
- Practices that are not used
- Frequently used practices that are perceived only helpful
- Good practices that are not used or only occasionally used

If these anomalies are found to be actual problems, there are usually many ways of solving the problems and improving the quality practices, e.g.

- Finding out new practices
- Adjusting or tailoring existing practices
- Creating new custom practices
- Enforce usage of the selected practices and improve tracking

3.1.2 Rhythm

Providing information on the achieved quality level is one of the core goals of testing and other quality practices. Short, time-boxed iterations provide a mechanism for rapid feedback during development and enable frequent control points in which the development plans can be revised. From the quality practices point of view these time-paced iterations and shorter heartbeat time horizon activities are essential mechanisms for managing quality practices as part of the tight development rhythm and for providing the quality information promptly[2]. In rapid time-paced development processes the quality practices are performed continuously on both heartbeat and iteration time horizons. Quality practices on heartbeat time horizon are performed and tracked continuously and on iteration time horizon the practices are performed in the context of each iteration. This

[2] Promptly here means that information is available in right time, i.e. early enough and when it is needed, and also that the information is provided frequently, i.e. as often as needed.

approach to quality assurance differs significantly from the traditional testing approach that is built on sequential development phases and V-model of software testing.

In most of the modern IID methodologies the quality practices and the whole quality assurance is rather poorly defined and described. Some methodologies leave the testing and quality assurance out and just state that the traditional quality assurance and testing approaches can be combined with iterative or even agile software development without problems. This, however, is far from the truth. Trying to combine traditional quality assurance with time paced or agile software development places significant challenges for the quality assurance. Instead, the quality practices must be planned to match the core principles of iterative software development and adjusted to the development rhythm of the particular development organization.

Using time horizons instead of traditional phased models makes it easier to understand the dynamic behavior of iterative and incremental development process. Using time horizons we can show how the quality practices are actually performed in IID context and how the quality information is provided promptly in the development rhythm.

An essential difference that can be used to categorize quality practices in IID is whether the practice is applied in sync with the development tasks or not. In-sync quality practices are practices that operate on the heartbeat time horizon and off-sync practices are synchronized to longer, iteration or release, time horizons.

In-Sync Quality Practices

In-sync practices are used to build quality into and to evaluate the achieved quality of a piece of functionality during its implementation work. In-sync practices apply to all implemented features or code, and the implementation tasks are not considered complete before these activities are performed.

A good example of an in-sync quality practice is automated unit testing; developers must write unit tests for each piece of code that they create and progress is taken into account only after the tests are completed and passed. Note, however, that in-sync practices are not time- boxed to the heartbeat time horizon – a task can take several heartbeats to complete, but tracking and controlling is performed with a constant rhythm by, e.g., heartbeat meetings and daily builds. Thus, these practices tightly follow the daily rhythm of the development activities. In-sync quality practices are not delayed to any later phase. Instead, these activities are performed as part of the other development tasks, regardless of whom, e.g., a developer or tester

performs these tasks. These activities give instant feedback to developers and therefore help drive development in the right direction. In agile methods, quality assurance on the heartbeat time horizon is very strong as can be seen e.g. in the practices of Extreme Programming (XP) which mostly are in-sync practices.

In-sync quality practices are not restricted to the testing tasks of the developers. In-sync practices can include, for example, designing and executing system level functional tests, executing regression tests, reporting test results and bugs, verifying bug fixes, and so on. Rhythm is the key: in-sync activities are managed and tracked according to the heartbeat rhythm and the different roles must communicate and synchronise their work in every heartbeat.

Off-Sync Quality Practices

Quality practices on the iteration time horizon are not performed for each individual implemented feature at the heartbeat rhythm. Instead, these activities are controlled and tracked on the iteration time horizon against the iteration (quality) goals. These kinds of quality practices are off-sync practices, because they are not tightly synchronized with the development tasks. Off-sync practices include, e.g., all testing and review activities that are needed to ensure that the end product of an iteration has good enough quality.

In practice, all required quality activities for new functionality and code cannot be done at the heartbeat rhythm, and it is not even desirable. Many activities are not directly connected to individual features or enhancement tasks that developers perform. This kind of QA activities can be tracked on the iteration time horizon by including them in iteration goals and tasks. Many tasks of professional testers are off-sync practices by nature. Specialised testers do a lot of testing, test case design, test environment setup, etc., which is not directly connected to the development tasks (e.g. testing performance, reliability, and other qualities on the system test level that cover the system broader than functional testing of single functions or function groups). These specialists write and execute tests during the whole iteration, and they must synchronise their work with the developers. The synchronisation can be done using code handoffs at the daily or weekly build rhythm, and the testers can, e.g., participate in development heartbeat meetings. In addition, testing tasks that require specific expertise or long periods of setup or execution time may best be managed as off-sync practices on the iteration or even release time horizon.

Iteration tasks are time-boxed, because the length of an iteration is fixed. During the iteration, testers have to prioritise their work. This means that it

is very important to track the progress of the work and communicate the situation to development leaders constantly, e.g., in heartbeat meetings. Otherwise, it is very easy to ignore the fact that off-sync quality practices lag behind development progress. This can lead to cutting corners on quality practices, which increases quality risks. The correct action would be to reduce the development scope of the iteration in order to get the required quality practices performed.

Some off-sync quality practices can be operated also on release time horizon. Such practices include evaluating the test results and other quality information from the individual iterations and practices to steer the development project based on that information (e.g. planning forthcoming iterations).

Rhythm Helps to Select and Understand Quality Practices

Identifying the rhythm of the quality practices helps understanding the nature of the practices and what kind of information the practices can provide and when. It also clarifies when the practices must be performed in incremental development process, which is not as clear and straightforward to follow as it does not consist of clearly separated phases.

The nature of quality practices with different rhythm is also different. The characteristics and differences of in-sync and off-sync practices are summarized in the Table 5.

Table 5 Characteristics of in-sync and off-sync quality practices

In-Sync Quality Practices	Off-Sync Quality Practices
Are always done before the task is considered completed	Are done by the iteration or release deadline
Provide prompt feedback Feedback is provided early Information is available when needed Information is provided frequently	Cannot provide prompt feedback Provide feedback in the rhythm of the time horizon in question Information can be utilized on longer time horizon
Goal is to build in quality and verify Verifying quality level of each task or feature	Goal is to evaluate achieved quality level and validate Demonstrating true progress
Are easy to track and measure	Are prioritized
Cannot be too large tasks	Can provide more thorough testing and analysis of the quality
Focus on individual units/components/features/tasks	Focus on larger entities and how things work together (whole system)
Revealed problems can be tackled right away Less risk is delayed to later phases	Higher risks associated with the results

Sometimes there is a need to have quality practices that are not connected to the short iterations of iterative software development process, or it just happens to be so. This kind of quality practices can be called no-synch practices, since they are not connected to the rhythm of the development

iterations. If the practices are not connected to any iteration goals, it is hard to plan and track the work. Such practices are risky and require special attention to ensure that the practices get done and the results are available when needed. It would be better to somehow connect all quality practices in iterations and their goals, or at least make them clearly visible in the release goals. Typical examples of such no-sych practices are testing that cannot be completed in the iteration schedule, tasks of a separate testing group, and testing in multiple environments. Also the common practice of including separate stabilisation iteration at the end of the release project is an example of a no-sync quality practice. This, however, means that certain quality risks are not revealed until the last iteration.

3.1.3 **Viewpoints**

The third part of the TQP Framework is viewpoints. The framework lists three viewpoints to quality practices: *Purpose*, *Attitude*, and *Roles*. Each quality practice can be viewed through these three viewpoints and the distributions of the quality practices among the dimensions of the viewpoints can be used to reveal soft points in the set of quality practices and to give guidance to improvements. Next we present the three viewpoints and their dimensions.

Table 6 Viewpoints of the TQP Framework

Purpose	Preventing Defects – Building in quality Acting early to ensure high quality	Detecting defects – Evaluating achieved quality Measuring and tracking trends
Attitude	Constructive – Demonstrating benefits Showing progress and building confidence	Destructive – Pointing out problems Reporting defects and information on risks
Roles	Developers – Are subjective Difficult to see faults in one's own software	Independent testers – Are objective Can take the viewpoint of customer and user

Purpose

The first viewpoint describes the characterizing purpose of a quality practice. Is the practice focused on *preventing defects* or *detecting defects?*

Quality practices that are focused to preventing defects are used affect the outcome of the development tasks before or during the tasks are performed. The idea is to strive for building in quality as part of the development work. The nature of these practices is to act early and support development work.

Practices that are focused to detecting defects are used to measure and evaluate the achievements, right after the task is done or in some later

control point. The idea is to evaluate the achieved quality level or track metrics. The nature of these tasks is to provide information and measures of the achieved quality level at the rhythm of certain time horizon.

Attitude

The second viewpoint is about the attitude of a quality practice. Is the practice *constructive* or *destructive* in nature?

Constructive practices are focused to demonstrating achieved benefits. These practices are used for showing progress or for building confidence in true achievements by demonstrating working software. However, there is a risk of relying too much on constructive practices, because people tend to see what they want to see. This means that if you want and expect a program to work, you are more likely to see a working program, making you miss defects, for example in testing.

Destructive practices, instead, focus to pointing out problems and defects. The idea of destructive practices is to help increasing the quality by revealing defects and making sure they get fixed.

Roles

The last viewpoint is roles. Quality practices can be performed by people in different roles and the traditional theory of software quality assurance emphasizes the importance of independent testing and QA. Is the quality practice *performed by developers* or is it *performed by independent testers?*

For developers it seems to be difficult to see problems in one's own code, for example, and, more importantly, developers own tests do not reveal possible misunderstandings of the specifications or requirements. It is important to recognize the effect of role in the results of quality practices.

Independent testers can see the tested software more objectively. There are many alternative roles regarding the level of independence. Here is a list of some possible roles from least to most independent role.

1. Developer of the code or feature
2. Another developer of the team
3. Tester as part of the development team
4. Independent tester or developer outside of the development team
5. Customer or the project or user of the developed product

Using TQP Framework for Balancing Quality Practices

In Appendix A, we present an example of using the TQP Framework to analyse different incremental software processes.

4 Root Cause Analysis

4.1 Introduction

The discipline of software engineering was born in 1968 due to problems in software projects (Naur, Randel 1969). Logically, preventing the problems of software projects is the key for software process improvement (SPI), which aims lowering the costs of development work, shortening the time to market, and improving the product quality (Burr, Owen 1996).

This chapter introduces you a lightweight root cause analysis method, referred to as the ARCA method (Lehtinen et al. 2011). The method is feasible for analyzing the causes of quality deviations, resulting to sensemaking and quality practice improvements. The ARCA method includes four steps: Problem Detection, Root Cause Detection, Corrective Action Innovation, and Documentation of the Results.

The ARCA method was piloted in four medium-sized software product companies, to find the causes of the target problems of the companies. We will introduce you the causes detected in these four software companies by utilizing a classification system presented in Appendix C. Additionally, we will present the feedback of the case participants and the required effort including the amount of causes detected and corrective actions developed in the cases.

Software engineering problems have been widely studied. Demir (Demir 2009) indicates that scope management, requirements management, estimation, and communication are the most usual areas of challenges in software project management. Kappelman, McKeeman, and Zhang (Kappelman et al. 2006) introduce a list of 53 early warning signs of IT project failure. Furthermore, Cerpa and Verner (Cerpa, Verner 2009) summarize a list of 18 factors for software project failure.

The key for effective problem prevention is to know why the problem occurs (Rooney, Vanden Heuvel 2004). The problems and events of software engineering are logically interconnected with a cause and effect relationship (Lehtinen et al. 2011), indicating that the problem is a cause for some other problem and vice versa. For example, the estimation problem

introduced by Demir (Demir 2009) may be caused by the problem of miscommunication introduced in the same study.

Root cause analysis (RCA) is a structured investigation of a problem to detect the causes that need to be prevented (Latino, Latino 2006). RCA takes the problem as an input and provides a set of its causes with cause and effect structure as an output (Lehtinen et al. 2011). It aims to state what the problem causes are and where they occur. This helps with software process improvement in various contexts and across all software organizations including product development, hardware design, product engineering, and manufacturing (Mays 1990). The RCA method consists of three general steps: target problem detection, root cause detection, and corrective action innovation (Lehtinen et al. 2011).

Considering quality deviations, most of the reported industrial cases in software engineering root cause analysis have aimed to lower defect rates by preventing the causes of the most typical types of the defects. The results are promising: a 50 percent decrease in defect rates (Card 1998), a 53 percent savings in costs and a 24 percent increase in productivity (Leszak et al. 2000) have been indicated. However, the high number of particular types of software defects is not the only quality deviation that should be analyzed; e.g., low functionality, low usability, low performance, and low installability are all industrially relevant and severe quality deviations worth of being processed further.

4.2 The ARCA Method

The ARCA method is a lightweight root cause analysis method feasible for analyzing the causes of the quality deviations of your project. The method includes four steps: Problem Detection, Root Cause Detection, Corrective Action Innovation, and Documentation of the Results. Problem Detection consists of selecting a target problem and collecting its preliminary causes, e.g., “why the product installation is highly difficult?” Root Cause Detection consists of a workshop session where the causes of the target problem are detected, analyzed and organized into a cause-effect diagram. Corrective Action Innovation consists of a workshop session, which is focused on developing corrective actions for the most important causes, e.g., what new quality practices are needed and how to improve the existing practices? Documentation of the Results consists of documenting the detected causes and corrective actions.

The ARCA method is performed by an RCA team which is lead by an RCA facilitator, e.g., the quality manager of your organization. The team consists of target problem experts including project managers, product managers,

developers, and testers. The team members are selected for both workshop sessions separately, because they might require different expertise.

Table 7 summarizes the steps and work phases of the ARCA method and its antecessors. We will introduce you these steps in the forthcoming sections by introducing the ARCA method in details while comparing it to the prior RCA methods.

Table 7 Summary of the ARCA and prior RCA methods and their work phases

RCA Method	Target Problem Detection Step		Root Cause Detection Step		Corrective Action Innovation Step	
	Work Phase	Work Practices	Work Phase	Work Practices	Work Phase	Work Practices
Rooney (Rooney, Vanden Heuvel 2004)	Data collection	Interviewing, inspections	Causal factor charting	Sequence diagram	Recommendation generation	-
			Root cause identification	Decision diagram		
Ammerman (Ammerman 1998)	Problem definition and data collection	-	Event and causal factor charting	Sequence diagrams	Corrective action development	Interviewing
	Task analysis	Paper-and-pencil, walk-through	Root cause determination	Interviewing, event and causal factor charts, lists, and worksheets		
	Change analysis	Flow charts				
	Control barrier analysis	Flow charts				
Latino (Latino, Latino 2006)	Opportunity analysis	Sequence diagrams, interviewing, Pareto analysis	Data analysis	Flow chart, logic tree, meetings	Recommendation development	Writing individually, meetings
Card (Card 1998)	Defect sampling	Sampling, meetings	Determining principal cause	A fishbone diagram, cause categories, meetings	Development of action proposals	Meetings
	Defect classification	Classification scheme, meetings				
	Identifying systematic errors	Pareto analysis, meetings				
ARCA (Lehtinen et al. 2011)	Target problem detection	A focus group meeting	Preliminary cause collection	Anonymous email inquiry, a directed graph	Root cause selection	Email inquiry
			Causal analysis workshop	Brainwriting and brainstorming in a meeting, a directed graph	Corrective action workshop	Brainwriting combined with skeptical and optimistic perspectives in a meeting

4.2.1 **ARCA Method Step 1: Target Problem Detection**

This is the first step of the ARCA method. After this step, the target problem will have been defined. In the ARCA method, the first step starts with a focus group meeting where the target problem is defined and the causal analysis workshop participants, who are to collect the target problem causes and to evaluate root causes, are selected (four to ten participants). The RCA facilitator holds this meeting with company staff, e.g., the managers who are responsible for product quality. In the meeting, the following issues should be justified and documented: what is the target problem and why exactly is this problem important to prevent? When selecting the causal analysis participants, it is important to include target problem experts that represent different stakeholders around the target problem. These may include project managers, developers, testers, software quality assurance staff, product managers, and process improvement group members.

Rooney (Rooney, Vanden Heuvel 2004) and Latino (Latino, Latino 2006) indicate that interviewing is a feasible practice in detecting the target problem. However, we emphasize a focus group meeting because it is an excellent approach to identify rapidly what is important to the people (Lethbridge et al. 2005). It requires less effort than interviewing, is easy to conduct, and combines the knowledge of people efficiently.
Problem sampling is unfeasible for many target problems. It sounds like a great idea to analyze and eliminate the causes of the most usual type of problems to lower the likelihood of their reoccurrence. On the other hand, problem sampling requires effort and information that is not easily available in practice (Wagner 2008) as the defect type or defect module is sporadically reported by the company's personnel (Mäntylä et al. 2011) making the defect data too unreliable for RCA. Moreover, the problem sampling can be done only for the problems that are reported (Kalinowski et al. 2008), and, in many cases, defect databases do not contain problems such as requirements faults (Gursimran, Jeffrey 2009). (Lehtinen et al. 2011)

4.2.2 ARCA Method Step 2: Root Cause Detection

This is the second step of the ARCA method. After this step, the most important root causes will have been detected and evaluated. In the ARCA method, the second step consists of two work phases: preliminary cause collection and a causal analysis workshop.

In the preliminary cause collection, the RCA facilitator sends out an email inquiry to the case participants and collects the target problem causes. The inquiry asks the participants to list at least five causes of the target problem. Since the listed causes probably complement one another, they are organized into a cause-effect diagram by the RCA facilitator, as presented in Figure 3. Using a software tool is recommended here.

The second work phase is the causal analysis workshop, which is prepared by the RCA facilitator. A cause entity (see the colored causes in Figure 3) includes a cause and its sub-causes, which together form an entity that is reasonable to process together. By analyzing the cause-effect diagram, the RCA facilitator selects the most important cause entities to be processed in the workshop. It is possible that the entities will overlap since the causes explain one another. Processing a cause entity containing about ten causes can be done adequately in about 40 minutes. We recommend this as a suitable size for a cause entity.

The causal analysis workshop is a meeting where new target problem causes are collected and analyzed. The workshop has a recommended minimum duration of 40 minutes per cause entity. At the beginning of the workshop, the RCA facilitator presents the target problem, the preliminary causes, and the selected cause entities. Thereafter, new causes are collected for each selected cause entity. The cause entities are processed one at a time. Each cause can either deepen or widen a cause entity. Collecting the causes into a cause entity is done in three parts:

1. The participants write new causes down on paper for five minutes (the cause-effect diagram should be projected onto the wall)
2. Each participant presents the causes he or she has written and explains where they should be placed in the cause-effect diagram
3. The participants briefly discuss the cause entity's causes, trying to brainstorm more causes and to recognize whether a cause has a relationship to other causes

After all the selected cause entities have been processed, the related cause-effect diagram is analyzed as a whole. The RCA facilitator asks the participants to point out essential causes and to discuss them. The

controllable causes, i.e., the root causes, are identified. The other causes are set aside and are not processed any further.

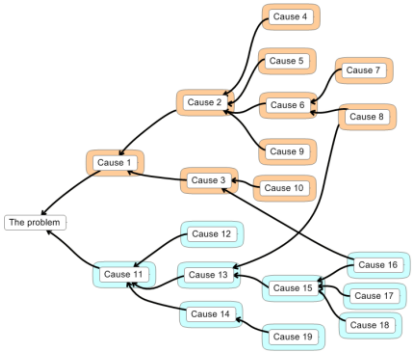

Figure 3 The cause-effect diagram of the ARCA method

Figure 4 Template for corrective actions

We see that both anonymous and public approaches are important in root cause detection. Anonymity encourages the participants to address causes that they believe are dangerous to say aloud, whereas publicity helps to address causes that many participants value highly. The other RCA methods do not emphasize this. Ammerman (Ammerman 1998) emphasizes interviewing only, whereas Latino (Latino, Latino 2006) and Card (Card 1998) emphasize meetings.

Unlike the prior RCA methods, we recommend using a directed graph (Björnson et al. 2009) to structure the causes based on their cause-and-effect relationships (see Figure 3). As the directed graph represents a network of causes, each cause needs to be placed only once in the cause-effect diagram. The cause-effect diagrams of the prior RCA methods result in the problem of duplicating the same cause multiple times if the cause simultaneously explains more than one cause. Card (Card 1998) recommends using a fishbone diagram, which he claims to be a simple technique. However, using the fishbone diagram does not solve the duplicating problem. The problem also occurs when a logic tree is used, which is recommended by Latino (Latino, Latino 2006). Rooney (Rooney, Vanden Heuvel 2004) recommends using a sequence diagram followed by a decision diagram. Unfortunately, the sequence diagram also includes the duplicating problem and the decision diagram includes the challenge of detecting the correct problem causes in advance, as the target problems vary. Additionally, we believe that using two diagrams is more difficult than using one. Ammerman (Ammerman 1998) indicates that structuring the target problem causes should be done with visual tools such as lists, worksheets, and charts. However, it is likely that too many target problem causes will be detected to be visualized using these tools (Jalote, Agrawal 2005). Additionally, the duplicating problem occurs with lists, worksheets, and charts. (Lehtinen et al. 2011)

4.2.3 ARCA Method Step 3: Corrective Action Innovation

This is the third step of the ARCA method. After this step, the corrective actions for the most important root causes will have been developed. In the ARCA method, the third step consists of two work phases: the root cause selection and the corrective action workshop. The first work phase includes the selection of the root causes. To focus the available resources as efficiently as possible, the RCA facilitator has to carefully select the root causes for which corrective actions are to be developed. First, the finalized cause-effect diagram is sent to the participants of the causal analysis workshop. The participants are asked to propose root causes for which corrective actions should be developed and evaluate them using the following criteria: the level of impact on the target problem and the level of difficulty of developing corrective actions. Then, the RCA facilitator selects four to six root causes to be processed using his judgment and analysis of the root causes proposed by the participants. Finally, the RCA facilitator

documents each of the selected root causes including its sub-causes into a cause-effect diagram, each for an individual paper.

The second work phase of the step is the corrective action workshop, which is a meeting wherein the corrective actions of the selected root causes are developed, evaluated, and analyzed. The workshop has a recommended duration of two hours. First, the RCA facilitator selects four to six participants to join the workshop. They have to be an aggregate of experts who are as competent as possible at solving the selected root causes. In the workshop, each participant works, in turn, for ten to fifteen minutes with one root cause. They develop corrective actions by writing them down on paper (see Figure 4) and rotating them through the participants. The root causes are rotated until every participant has treated all the root causes. A participant can also supplement corrective actions developed by other participants by adjusting, expanding, and commenting on them. The corrective actions are evaluated to find the best corrective actions. The evaluation is conducted similarly to their development: the root causes, including their corrective actions, are rotated through the participants. Each participant evaluates corrective actions of a root cause by giving two attributes to each (scale of one to five): impact on the target problem and feasibility. The last participant evaluating the corrective actions of a root cause calculates the sum of evaluations of each corrective action. Then, he presents the corrective action that has the highest value of the multiplication of the impact and feasibility. This is done for each processed root cause. The participants are asked to discuss the corrective action and to refine it. The presenter writes down the comments and improvement suggestions concerning the action he presented.

> In the prior RCA methods, there is very little practical guidance on how to develop corrective actions. Keeping a meeting where the corrective actions are developed is presented by Latino (Latino, Latino 2006) and Card (Card 1998), whereas Ammerman presents interviewing techniques to be used (Ammerman 1998). We believe that keeping the meeting helps to develop commitment to the corrective actions among the participants more than the interviewing techniques. In the corrective action innovation, we chiefly emphasize brainwriting because it provides an efficient way to use all of the participants simultaneously. However, as there are also advantages in brainstorming, we recommend it to refine the findings into the best corrective actions. Latino emphasizes brainstorming in the corrective action innovation but stresses also that it is important to write down the corrective actions (Latino, Latino 2006). Ammerman indicates that it is important to develop multiple corrective actions and to evaluate and select them to have alternatives (Ammerman 1998). We found that the commonality between the elimination techniques presented in the literature (Andersen, Fagerhaug 2006) is that a corrective action is analyzed from different perspectives, especially from optimistic and skeptical perspectives. Therefore, we adopted the idea of different perspectives to the ARCA method by creating a paper template for a corrective action that forces the participants to brainwrite the corrective actions from both perspectives. (Lehtinen et al. 2011)

4.2.4 **ARCA Method Step 4: Documentation of the Results**

During this final step of the ARCA method, the results are compiled into a final report, which includes at least the target problem definition, the cause-effect diagram, and all of the corrective actions, including their evaluations. This step is also mentioned by Card (Card 1998), Latino

(Latino, Latino 2006), and Ammerman (Ammerman 1998). The best corrective actions should be implemented to make the actual changes in the way of working. Because gaining currency for a corrective action can be challenging, the final report can be used to justify the changes required to prevent the target problem. Additionally, the final report can be a valuable source of cause information in future RCA cases. (Lehtinen et al. 2011)

4.3 Industrial Cases

The ARCA method was piloted at four software companies. Table 8 summarizes the company cases with the data that is important for using the ARCA method. Including the effort the company has expended trying to solve their target problem previously, the table summarizes how the key representatives characterized the target problems and how the case participants evaluated it.

Table 8 Summary of the case contexts

	Case 1	**Case 2**	**Case 3**	**Case 4**
Case company	Software company with 100 employees	Software company with 450 employees	Software company with 100 employees	Software company with 110 employees
Target problem	Fixing and verifying defects delays project schedules	Blocker type defects are detected in the product after releases	New product installation and updating are challenging tasks	Issues' lead time is sometimes intolerably long
Roles of the case participants	Project managers, quality managers, developers, sales personnel, N = 9	Mostly developers, N = 9	Project managers, testers, developers, N = 7	Project managers, testers, developers, sales personnel, N = 6
Target problem characteristics	*"Extremely costly and complex"*	*"Not very costly, but very complex"*	*"High impact on customer relationships and complex"*	*"Extremely costly and complex"*
Difficulty of preventing the target problem *	Average = 5.3 Standard deviation = 1.1	Average = 5.6 Standard deviation = 0.8	Average = 5.4 Standard deviation = 1.3	Average = 5.5 Standard deviation = 1.0
Earlier effort surrounding the target problem *	*"We have continuously tried to solve this"* Average = 3.0 Standard deviation = 1.0	*"During recent months, we have reacted to this"* Average = 4.3 Standard deviation = 1.3	*"We haven't managed this much"* Average = 3.4 Standard deviation = 0.5	*"We have discussed how to improve communication"* Average = 3.0 Standard deviation = 0.6
Impact of the target problem *	Average = 5.8 Standard deviation = 1.1	Average = 5.0 Standard deviation = 1.3	Average = 5.6 Standard deviation = 0.9	Average = 5.9 Standard deviation = 0.9

*Scale: 1=very low; 2, 3, 4=neutral; 5, 6, 7=very high

4.3.1 **Case 1**

The first case was conducted at Company 1, a medium-sized international software product company with approximately 100 employees. The average size of the project organization is about seven people. The main product is a large and complex software system, released twice a year, consisting of a major and a minor release.

The target problem of the case was that the product releases are delayed due to a high number of software defects detected at the end of the development projects. The company has continuously tried to prevent the problem during recent years. The key representatives' common opinion was that the problem is extremely complex and costly for the company. They claimed that the main problem causes are that the size of technical blocks in

the software is too large and that employees' attitudes are not fertile enough to develop high-quality software at once. Additionally, they assumed that increasing discipline among the developers and releasing the software in shorter cycles would help in eliminating the target problem.

Table 9 introduces the distribution of problem causes at Case 1. The highest number of causes was focused on the process area of software testing including 37.7% of causes. These causes were widely related to lack of resources, bad values & responsibility taking, lack of instructions & experiences, and wrong work practices.

Table 9 Distributions of processed problem causes at Case 1 (130 causes, 162 edges, 40 blocks)

C%=percent of causes, E%=percent of edges to

C1	MA		S&R		IM		ST		PD		UN		Tot	
	C%	E%	C%	E%	C%	E%	C%	E%	C%	E%	C%	E%	C%	E%
Instructions & Exp. (P)	5.4	4.3	3.1	0.0	2.3	0.0	4.6	1.2	0.8	0.0	1.5	0.6	17.7	6.2
Work Practices (M)	3.1	1.9	1.5	2.5	2.3	0.6	4.6	2.5					11.5	7.4
Task Output (T)	7.7	7.4	1.5	9.9	2.3	0.0	3.1	11.1					14.6	28.4
Task Difficulty (T)	0.8	0.0	0.8	0.0			3.8	4.3	0.8	0.0			6.2	4.3
Existing Product (E)			0.8	0.0			2.3	0.6					3.1	0.6
Resources & Sch. (E)			0.8	0.0	3.0	5.0	6.2	4.9					10.0	9.9
Values and &Resp. (P)	6.2	7.4	0.8	0.6	2.3	1.2	5.4	7.4					14.6	16.7
Process (M)					1.6	0.6	3.1	5.6					4.6	6.2
Company Policies (P)			1.5	0.0	0.8	0.0	1.5	0.0					3.8	0.0
Co-operation (P)					1.5	0.6							1.5	0.6
Customers & Users (E)			0.8	0.0									0.8	0.0
Tools (E)	0.8	0.0											0.8	0.0
Task Priority (T)					5.4	8.0	1.5	4.3			0.8	0.0	7.7	12.3
Monitoring (M)					1.5	1.2	1.5	6.2					3.1	7.4
Tot	23.8	21.0	11.5	13.0	23.0	17.3	37.7	48.1	1.5	0.0	2.3	0.6	100	100

4.3.2 **Case 2**

The second case was conducted at Company 2, a medium-sized international software product company with approximately 450 employees. The company releases new software versions regularly and its products can be characterized as complex and model-based software.

The target problem of the case was that blocker-type defects are detected after the product releases, which increases the costs of re-development and irritates the users. The company has recently reacted to this problem by setting a clear goal to lower the number of defects detected by the customers. The key representatives characterized the target problem as very complex and including many different causes. The main causes for the target problem were believed to be the fact that new code is built on the old, low-quality code, too many different methods are used in the development work, and the lack of different hardware set-ups decreases the coverage of the software testing. They said that the problem could be best eliminated by refactoring the old code. They also believed that the problem is not very severe because the customers are currently highly satisfied.

Table 10 introduces the distribution of problem causes at Case 2. The highest number of causes was related to the causes of software testing practices (8.1%). The process area of software testing (ST) was additionally underlined most often (38.4%). The people stressed that the output of software testing is insufficient (6.5%) and that the software testing is difficult (6.5%). Additionally, the existing product makes the software testing even more challenging task (4.3%). It can also be seen that the case participants claimed the output of sales & requirements (S&R) being insufficient (4.3%).

Table 10 Distributions of processed problem causes at Case 2 (185 causes, 195 edges, 42 blocks)

C%=percent of causes, E%=percent of edges to

C2	MA		S&R		IM		ST		PD		UN		Tot	
	C%	E%	C%	E%	C%	E%	C%	E%	C%	E%	C%	E%	C%	E%
Instructions & Exp. (P)			2.7	1.0	4.9	8.7	3.8	6.2					11.4	15.9
Work Practices (M)			1.1	0.0	3.2	1.0	8.1	12.8	0.5	0.0			13.0	13.8
Task Output (T)	1.6	0.5	4.3	6.2	4.9	6.2	6.5	9.7	0.5	0.0			17.8	22.6
Task Difficulty (T)			1.6	1.0	1.6	1.0	6.5	5.1					9.7	7.2
Existing Product (E)			1.1	0.5	4.3	7.2	4.3	2.1	0.5	0.0			10.3	9.7
Resources & Sch. (E)					3.2	1.5	2.2	0.5	1.6	4.1			7.0	6.2
Values and &Resp. (P)	1.1	0.5	2.2	1.0	3.2	1.5	2.2	2.1			1.0	0.5	9.6	5.6
Process (M)					2.7	2.0	2.2	1.0	1.1	0.0			5.9	3.1
Company Policies (P)			0.5	0.0	1.6	1.5	0.5	0.5	0.5	1.5			3.2	3.6
Co-operation (P)			1.1	2.6	2.2	1.5							3.2	4.1
Customers & Users (E)			3.8	5.1	1.0	0.0							4.9	5.1
Tools (E)			1.1	0.0			1.1	0.0					2.2	0.0
Task Priority (T)					1.0	2.5							1.1	2.6
Monitoring (M)							1.1	0.5					1.1	0.5
Tot	2.7	1.0	19.5	17.4	34.0	34.9	38.4	40.5	4.9	5.6	1.0	0.5	100	100

4.3.3 **Case 3**

The third case was conducted at Company 3, a medium-sized international software product company with approximately 100 employees. The main product can be characterized as a highly configurable software service. The product is delivered for the customers through installation projects that occasionally include the development of new features. New software versions are released regularly.

The target problem of the case was that the installation projects are too challenging to be performed efficiently. It often follows that re-engineering has to be done because of unexpected defects caused by the complex software configurations and new development work during the projects. The company hasn't expended much effort to manage the target problem earlier. However, the key representatives stressed that the target problem has a significant impact on their customer relationships and that it is very complex to prevent. They said that the main cause of the target problem is that the employees have too many different ways in which to perform a product installation. Additionally, the number of different stakeholders is too high with respect to the quality of communication between them. They

also indicated that the target problem could be minimized by creating checklists and simplifying the installation process.

Table 11 introduces the distribution of problem causes at Case 3. The process area of product deployment (PD) included the highest number of causes (52.4%). Lack of instructions & experiences to conduct the product deployment was the most usual cause (14.7%). Also the work practices of product deployment was underlined many times (10.5%). In contrast to other cases, a high number of causes related to the causes of existing product (9.1%) were detected in this case.

Table 11 Distributions of processed problem causes at Case 3 (143 causes, 144 edges, 34 blocks)

C%=percent of causes, E%=percent of edges to

C3	MA		S&R		IM		ST		PD		UN		Tot	
	C%	E%	C%	E%	C%	E%	C%	E%	C%	E%	C%	E%	C%	E%
Instructions & Exp. (P)	0.7	0.0			2.8	2.1	3.5	1.4	14.7	9.7			21.7	13.2
Work Practices (M)	0.7	0.0			4.9	2.8	2.1	1.4	10.5	7.6			18.2	11.8
Task Output (T)	4.2	1.4			5.6	2.8	3.5	5.6	4.2	14.6			17.5	24.3
Task Difficulty (T)	0.7	0.0					2.8	6.9	6.3	21.5			9.8	28.5
Existing Product (E)					2.1	3.5	0.7	0.7	9.1	8.3			11.9	12.5
Resources & Sch. (E)	0.7	1.4					1.4	0.7	2.1	0.0			4.2	2.1
Values and &Resp. (P)														
Process (M)					1.4	0.7	1.4	1.4	0.7	0.0			3.5	2.1
Company Policies (P)	0.7	0.0			0.7	2.1							1.4	2.1
Co-operation (P)														
Customers & Users (E)	0.7	0.0			2.1	2.1	2.1	0.0	4.2	0.7			9.1	2.8
Tools (E)					0.7	0.0	0.7	0.0					1.4	0.0
Task Priority (T)					0.7	0.7							0.7	0.7
Monitoring (M)									0.7	0.0			0.7	0.0
Tot	8.4	2.8			21.0	16.6	18.2	18.1	52.4	62.5			100	100

4.3.4 **Case 4**

The fourth case was conducted at Company 4, a medium-sized international software product company with approximately 110 employees. The main product can be characterized as a highly complex software system. The product is delivered to customers through complex integration projects where the product is configured into the software systems of the customers.

The target problem of the case was that the lead time of an issue is occasionally intolerably long, resulting in delays in projects. The company hasn't expended much effort to manage the target problem earlier. However, they have tried to improve communication between the stakeholders of the company. The key representatives valued the target problem as high because it has a severe financial impact. It follows that the projects are not finalized on time. They said that the main causes of the target problem are lack of communication between the stakeholders and the way the company is dividing resources between the issues. Usually, an issue with fairly low priority doesn't get enough resources. They concluded that preventing the target problem is not an easy task. This would require

increasing face-to-face meetings, increasing the number of inspections, and allocating skilled project managers to be responsible for the issues.

Table 12 introduces the distribution of problem causes at Case 4. It seems the problem of Case 4 was most often interconnected to the causes of management (MA), sales & requirements (S&R), and implementation work (IM). The process areas of sales & requirements (36.0%) and implementation work (31.0%) included the highest number of causes. The case participants claimed the work practices of implementation work (7.6%) and the instructions and experiences in sales & requirements (7.6%) the most. They also underlined a high number of causes of resources & schedules in both of these process areas and that there the practitioners should take more responsibility of their own work. Considering the causes of management, the case participants stressed the output (5.2%) and work practices (6.4%).

Table 12 Distributions of processed problem causes in Case 4 (172 causes, 252 edges, 40 blocks)

C%=percent of causes, E%=percent of edges to

C4	MA		S&R		IM		ST		PD		UN		Tot	
	C%	E%	C%	E%	C%	E%	C%	E%	C%	E%	C%	E%	C%	E%
Instructions & Exp. (P)	2.9	2.8	7.6	4.4	3.5	4.4	0.6	0.4					14.5	11.9
Work Practices (M)	6.4	9.9	5.2	11.5	7.6	9.5			1.2	0.8			20.3	31.7
Task Output (T)	5.2	4.4	3.5	4.0	3.5	2.4	1.2	0.0	0.6	0.8			14.0	11.5
Task Difficulty (T)	0.6	0.4	1.2	3.2	1.2	0.8	0.6	0.0			1.2	0.8	4.7	5.2
Existing Product (E)	1.2	0.4	1.7	0.0									2.9	0.4
Resources & Sch. (E)	1.7	0.4	5.2	4.8	5.2	7.5	1.2	2.8	0.6	0.8			14.0	16.3
Values and &Resp. (P)	1.7	1.6	5.2	6.7	3.5	1.6							10.5	9.9
Process (M)					4.0	2.8							4.1	2.8
Company Policies (P)			2.9	1.6	0.6	0.8							3.5	2.4
Co-operation (P)			1.2	1.6	1.7	4.0							2.9	5.6
Customers & Users (E)	0.6	0.0	2.3	0.0	1.2	0.0							4.1	0.0
Tools (E)	0.6	0.0			0.6	0.0							1.2	0.0
Task Priority (T)					2.9	2.0							2.9	2.0
Monitoring (M)					0.6	0.5							0.6	0.4
Tot	20.9	19.8	36.0	37.7	36.1	36.1	3.5	3.2	2.3	2.4	1.2	0.8	100.0	100.0

4.3.5 **Effort Used**

Table 13 presents the effort used and the number of case participants throughout the different steps of the ARCA method. In total, 73 to 98 man-hours were required to conduct the cases. The required hours were mostly dependent on the number of case participants because both workshop sessions were time-boxed. The effort used increases with each additional case participant.

Roughly a quarter of the total effort was used in RCA facilitator-specific activities, whereas the rest was used in activities that included the case participants (see Table 13). An average of 10 hours were used in step 1 (problem detection), 37 hours were used in step 2 (root cause detection), 25 hours were used in step 3 (corrective action innovation), and 12 hours were used in step 4 (documentation of the results).

Table 13 Effort used in the cases

(h=hours) and the number of case participants (n) (*= RCA facilitator only)

The step of the ARCA method		Case 1		Case 2		Case 3		Case 4		Avg		Std	
		h	n	h	n	h	n	h	N	h	n	h	n
S1	Problem definition meetings (startup)	17	10	10	5	6	6	6	4	9.6	6.3	5.3	2.6
S2	Preliminary cause collection (email inquiry)	3	7	5	5	3	6	1	4	3.2	5.5	1.5	1.3
	Organizing the cause-effect diagram (*)	9	1	10	1	17	2	9	1	11.3	1.3	3.9	0.5
	Causal analysis workshop	21	10	20	10	22	8	14	7	19.3	8.8	3.6	1.5
	Smartening up the cause-effect diagram (*)	4	1	4	1	4	1	4	1	4.0	1.0	0	0
S3	Root cause selection	6	5	6	8	3	6	5	7	5.2	6.5	1.5	1.3
	Corrective action workshop	23	8	24	11	18	8	16	7	20.3	8.5	3.9	1.7
S4	Final report (*)	12	1	12	1	12	1	12	1	12.0	1.0	0	0
Tot		98		96		90		73		89.3		11.4	

4.3.6 **Output of the Method**

Table **14** presents the results of the method in the cases. The target problem causes were detected by the preliminary cause collection (52 to 108 causes) and by the causal analysis workshop (80 to 137 causes). The effort used was not fixed in the preliminary cause collection, whereas it was fixed in the causal analysis workshop.

A total of two to six root causes were selected in the cases. Together with their sub-root causes, the selected root causes formed a set of root causes that was processed in the corrective action workshop. In each case, 24 to 77 root causes were processed and 13 to 40 corrective actions were developed. The processed root causes covered 10% to 45% of the total number of the detected target problem causes in each case (average = 25%).

Table 14 Results of the method

		Case 1	**Case 2**	**Case 3**	**Case 4**	**Avg**	**Std**
Step 2	Target problem causes from the preliminary cause collection	93	108	66	52	80	25.4
	Target problem causes from the causal analysis workshop	80	137	105	116	110	23.7
Step 3	The number of selected root causes	6	2	5	6	5	1.9
	The number of processed root causes, including sub-root causes	41	24	77	42	46	22.3
	The number of corrective actions	38	13	33	40	31	12.4

High-quality corrective action is highly feasible and equally effective.

Figure **5** presents the impact and feasibility of the corrective actions per case as a scatter chart. In each case, every case participant evaluated the impact and feasibility of each corrective action to detect the highest-quality corrective actions. The evaluations were done using a numerical scale, comprised of integers between one and five. We calculated the averages of the evaluations for each corrective action. The corrective action that had the highest value of the multiplication between the average impact and the average feasibility was interpreted as the highest-quality corrective action.

It is interesting that the proportion of the high-impact (avg. >= 3) corrective actions was larger than the proportion of the low-impact (avg. <

3) corrective actions in each case. Instead, the proportion of the high-feasibility (avg. >= 3) corrective actions was larger than the proportion of the low-feasibility (avg. < 3) corrective actions only in cases 1 and 4. It seems to be easier to develop high-impact corrective actions than to make them feasible.

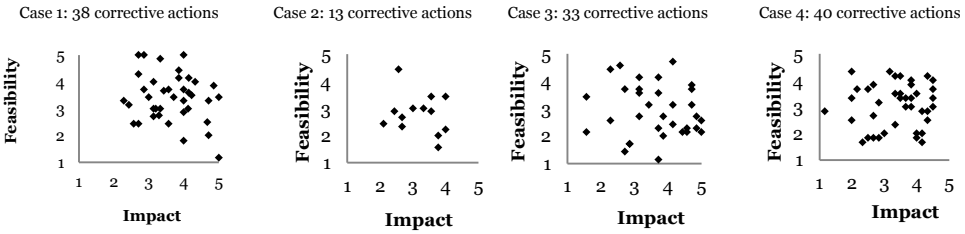

Figure 5. Corrective actions of the cases (Scales: 1=low; 2, 3, 4, 5=high)

4.3.7 **Feedback of the Case Attendees**

Table 15 summarizes the data from the query forms after the causal analysis and corrective action workshops. There, the steps of the root cause detection and the corrective action innovation are presented from three different perspectives. The first perspective is the easiness of the method. The second perspective is the usefulness of the method. The third perspective emphasizes the quality of the outputs of the ARCA method, including the comparison of the method to the current process improvement practices of the case companies.

The case participants experienced the corrective action innovation step as highly easy to use (avg. = 5.9), whereas the step of the root cause detection was experienced as only slightly easy to use (avg. = 4.7). The participants experienced that both of these steps are useful. They also experienced that correct target problem causes were detected and that fairly feasible corrective actions that have a high impact on the target problem were developed. The communication in both steps was experienced as highly explicit. The impact of the corrective actions was evaluated to be generally higher than their feasibility. The case participants experienced that the processed root causes were important for both product quality and the target problem. Unfortunately, the evaluation of the importance of the processed root causes for the target problem was done only in cases 3 and 4. The case participants experienced the root cause detection step as a more effective method to detect new process improvement opportunities than their current process improvement practices (avg. = 5.2). Similarly, the corrective action innovation step was experienced as a more effective method to develop process improvement ideas (avg. 6.1).

Table 15 Feedback of the case participants

(N=the number of respondents, Avg=average, Std=standard deviation, scale: 1=very low; 2, 3, 4=neutral; 5, 6, 7=very high)

	Case 1			Case 2			Case 3			Case 4		
	N	Avg	Std	N	Avg	Std	N	Avg	Std	N	Avg	Std
Root Cause Detection												
Easiness	9	4.3	0.8	7	4.9	1.2	6	5.1	1.2	6	4.8	0.4
Usefulness	9	5.4	0.4	8	5.8	0.6	6	5.8	0.5	6	5.3	1.0
Correctness of detected causes	8	6.0	0.5	8	5.8	0.7	6	6.2	0.8	6	5.5	0.8
Openness in communication	9	5.9	0.9	9	6.2	0.7	6	6.7	0.8	6	6.2	0.4
Efficiency comparison to company practices	9	5.4	1.2	8	5.1	1.0	6	5.3	1.2	6	4.8	1.3
Corrective Action Innovation												
Easiness	7	5.7	1.0	10	6.0	0.8	7	6.0	0.6	6	6.0	0.6
Usefulness	6	4.8	1.0	8	5.0	1.1	7	5.1	1.1	6	5.0	0.6
Impact of the CAs	7	5.6	0.5	9	5.4	0.7	7	5.9	0.7	6	5.3	0.8
Feasibility of the CAs	7	5.3	0.5	10	4.4	1.1	7	5.3	0.8	6	5.7	0.8
Importance of processed causes for target problem	-	-	-	-	-	-	7	5.7	0.8	6	5.3	0.8
Importance of processed causes for product quality	7	5.4	0.5	10	5.6	0.8	7	6.3	0.8	6	5.3	0.5
Openness in communication	6	6.5	0.5	9	6.1	1.3	7	6.1	0.9	6	6.3	1.2
Efficiency comparison to company practices	6	6.2	1.0	8	6.0	0.9	7	6.1	0.7	6	6.3	0.5

Table 16 summarizes the answers of the key representatives when they were interviewed after the cases. Our goal was to evaluate how they experienced the easiness and usefulness of the ARCA method and to include the effort used with respect to the output of the method under the evaluation.

Table 16 Interviews of the key representatives

Question	Case 1	Case 2	Case 3		Case 4
	Person 1	Person 2	Person 3a	Person 3b	Person 4
How easy and learnable is the method?	*"Easy to use and internalize."*	*"Easy in contrast to required effort and the output of the method."*	*"Easy to use and learn."*	*"It is fairly easy to use and learn. Organizing the causes was challenging."*	*"It was easy with the assistance of the researchers."*
Were the detected root causes significant with respect to the target problem?	*"Most of the causes were significant."*	*"As a general rule, yes. We have already reacted in one of the causes."*	*"Yes, they were. They matched well with my conception."*	*"Yes they were. I already knew some of those."*	*"Yes they were. The causes were mainly issues that lead the problem."*
Do the corrective actions prevent the target problem?	*"Yes, I think they do because they have a major impact on the processed root causes."*	*"No, I think that the corrective actions don't prevent the problem, but they do help us to improve our processes."*	*"Yes they do. We wouldn't even need to implement them all."*	*"I think that the corrective actions won't remove the problem completely, but they do have a major impact on the problem's sub-fields."*	*"Yes, the impact would be enormous."*
Would it have been possible to get the same results at lower costs using some other method?	*"No. We wouldn't be able to get this many relevant corrective actions."*	*"The method didn't require much effort. However, there should be only one workshop session and I would drop the email inquiry."*	*"I don't believe that. I don't know any such method."*	*"I think that 'better practice' would mean smaller group size and more talented experts in the second workshop."*	*"Maybe some other brainwriting method, where ideas are developed in literal form, could work as well."*
Should your company adopt the method?	*"Yes, we should. This works."*	*"Maybe, because this is an easy method with much potential. Additionally, the costs are low."*	*"I think we should adopt this method."*	*"I would gladly try this method again. Formal prioritization was nice."*	*"We should use this method, or at least a very similar one."*

In general, it seems that the method was experienced as easy to use. On the other hand, organizing the causes was noted to be challenging (person 3b) and the assistance of the researchers made the method unnaturally easy to use (person 4). The key representatives' unanimous opinion was that their companies should adopt the method and that the results were experienced as beneficial in contrast to the effort used. Additionally, they were not able to name any other method that could reach equally advantageous results with lower costs than the ARCA method. They experienced that significant root causes were detected with respect to the

target problem, and most of them stressed that, if implemented, the developed corrective actions would have a high impact in preventing the target problems. As an exception, it was noted that the corrective actions don't prevent the target problem, but they do help the company to improve their processes (person 2).

References

Ammerman, M. 1998, *The Root Cause Analysis Handbook: A Simplified Approach to Identifying, Correcting, and Reporting Workplace Errors*, First Edition edn, Productivity Press, 444 Park Avenue South, Suite 604, New York, NY 1016, USA.

Björnson, F.O., Wang, A.I. & Arisholm, E. 2009, "Improving the effectiveness of root cause analysis in post mortem analysis: A controlled experiment", *Information and Software Technology*, vol. 51, no. 1, pp. 150 - 161.

Burr, A. & Owen, M. (eds) 1996, *Statistical Methods for Software Quality: Using Metrics for Process Improvement*, First Edition edn, ITP A division of International Thomson Publishing Inc.

Card, D.N. 1998, "Learning from our mistakes with defect causal analysis", *IEEE Software*, vol. 15, no. 1, pp. 56-63.

Cerpa, N. & Verner, J.M. 2009, "Why Did Your Project Fail?", *Communications of the ACM*, vol. 52, no. 12, pp. 130-134.

Chillarege, R., Bhandari, I.S., Chaar, J.K., Halliday, M.J., Moebus, D.S., Ray, B.K. & Wong, M. 1992, "Orthogonal Defect Classification - A Concept for In-Process Measurements", *IEEE Transactions on Software Engineering*, vol. 18, no. 11, pp. 943 - 956.
Demir, K.A. 2009, "A Survey on Challenges of Software Project Management", *Proceedings of the 2009 International Conference on Software Engineering Research Practice*, pp. 13-16.

Grady, R.B. 1996, "Software Failure Analysis for High-Return Process Improvement Decisions", *Hewlett-Packard Journal*, vol. 47, no. 4, pp. 15 - 25.

Gursimran, S.W. & Jeffrey, C.C. 2009, "A systematic literature review to identify and classify software requirement errors", *Information and Software Technology*, vol. 51, no. 7, pp. 1087-1109.

Jalote, P. & Agrawal, N. 2005, "Using defect analysis feedback for improving quality and productivity in iterative software development", *Proceedings of the Information Science and Communications Technology (ICICT 2005)*Infosys Technologies Limited Electronics City, Hosur Road, Bangalore, India, pp. 701 - 714.

Kalinowski, M., Travassos, G.H. & Card, D.N. 2008, " Towards a defect prevention based process improvement approach", *Proceedings of the 34th EUROMICRO Conference on Software Engineering and Advanced Applications*IEEE Computer Society, , September 3-5, pp. 199 - 206.

Kappelman, L.A., McKeeman, R. & Zhang, L. 2006, "Early Warning Signs of IT Project Failure: The Dominant Dozen", *Information System Management*, vol. 23, no. 4, pp. 31-36.

Latino, R.J. & Latino, K.C. (eds) 2006, *Root Cause Analysis: Improving Performance for Bottom-Line Results*, Third Edition edn, CRC Press, 6000 Broken Sound Parkway NW, Suite 300 Boca Raton, FL 33487-2742.

Lehtinen, T.O.A., Mäntylä, M.V. & Vanhanen, J. 2011, "Development and evaluation of a lightweight root cause analysis method (ARCA method) – Field studies at four software companies", *Information and Software Technology*, vol. 53, no. 10, pp. 1045-1061.

Leszak, M., Perry, D.E. & Stoll, D. 2000, "A Case Study in Root Cause Defect Analysis", *Proceedings of the 2000 International Conference on Software Engineering*Institute of Electrical and Electronics Engineers, Los Alamitos, CA, United States, pp. 428 - 437.

Lethbridge, T.C., Elliott Sim, S. & Singer, J. 2005, "Studying Software Engineers: Data Collection Techniques for Software Field Studies", *Empirical Software Engineering*, vol. 10, no. 3, pp. 311-341.

Mäntylä, M.V., Itkonen, J. & Iivonen, J. 2011, "Who Tested My Software? Testing as an Organizationally Cross-Cutting Activity", *Submitted for Software Quality Journal*, .

Mays, R.G. 1990, "Applications of Defect Prevention in Software Development", *IEEE Journal on Selected Areas in Communications*, vol. 8, no. 2, pp. 164-168.

Naur, P. & Randel, B. 1969, "Software Engineering: A Report on a Conference Sponsored by the NATO Science Committee", .

Rooney, J.J. & Vanden Heuvel, L.N. 2004, "Root cause analysis for beginners", *Quality Progress*, vol. 37, no. 7, pp. 45 - 53.

Wagner, S. 2008, "Defect Classification and Defect Types Revisited", *Proceedings of the 2008 workshop on Defects in large software systems (DEFECTS '08)*ACM, New York, pp. 39-40.

5 Intelligent Manual Testing

5.1 Introduction

Traditionally, work on software testing by both academic researchers and software development practitioners has focused on test automation, optimization, testing tools, and automatic test data generation. The hype around Agile software development has further increased the emphasis on automation, especially at the unit level. This strong emphasis on automation, however, does not mean that the role of skilled and experienced manual testers has somehow diminished or that the importance of testing performed by humans, instead of computers, has decreased. In fact, the role of manual testing on many application domains is as strong as ever. The improvements in test automation do not necessarily mean that the need for manual testing would decrease; manual testing is still the most important method for revealing new defects in new or modified code. Instead, test automation increases the quality of the software and removes repetitive and laborious tasks from manual testers, which makes testers' work less boring and more effective, motivating, and rewarding.

While test automation can help detecting and removing low level coding defects efficiently, manual testing can focus on revealing more difficult and demanding problems and validating that the developed software actually fulfills *customer's and user's needs and expectations*. Automation is good, e.g., for repetitive regression testing, low level technical verification, load testing, and performance testing. Human testers are good at recognizing a variety of different types of problems when they appear, without prior definition of the possible failures to expect. Human testers can assess how well the software fulfills the actual requirements and needs of the customer, and by manual testing one can, in practice, test how the system manages to perform the actual work it was designed for. A human tester can have such a deep understanding of the application domain and the work that the system is used for that it can never be implemented as automated tests. A

human tester can analyze complicated and realistic scenarios paying attention to the working of the system as a whole. This is why manual testing, especially manual testing performed by domain area experts, is still an important in software companies.

One of the primary goals of software testing is to reveal new, yet undiscovered defects. In addition to this fundamental goal, software testing usually includes some other objectives, e.g., measuring and providing confidence in the achieved level of quality, being able to demonstrate test coverage and show what activities have been carried out for assuring the quality of a software product. However, the goal of revealing defects, and getting them fixed, is a truly fundamental goal of testing, since if the serious defects are not revealed, there is little value of being able to present documentation of the performed testing activities, or building (false) confidence in the quality.

5.2 An Experience Based and Exploratory Approach to Testing

In practice, test execution is not a simple mechanic task of executing completely specified test cases, which can be easily carried out by a novice employee, outsourced, or even completely automated. Instead, testers' skills and knowledge are likely to be important in all phases and activities of testing, also during test execution. Indeed, often testers use test cases primarily as a means of structuring and guiding their work. Practitioner literature has acknowledged for over a decade the testing approach that does not rely on predesigned test cases, i.e., exploratory testing (ET). Reports on exploratory testing have proposed that ET, in some situations, could be even orders of magnitude more efficient than test case based testing. Other claimed benefits of ET include the ability to better utilize testers' creativity, experience and skills, lower documentation overhead and lower reliance on comprehensive documentation.

5.2.1 Test Case Design and Documentation

Testing in the software engineering literature is considered a process based upon the design, generation, selection, and optimization of a set of test cases for testing a certain system or functionality. Various methods and techniques that aim determining and designing good or optimal test cases have then been developed. The underlying assumption is that given the right set of documented test cases, testing goals can be achieved by more or less mechanically executing the test cases. However, this view is problematic for at least three reasons. First, empirical studies of testing

techniques show that there are many other factors than the technique used to design or select test cases that explain the effectiveness and efficiency of testing. These include, among others, properties of the actual software being tested, the types of the actual defects in the tested software and the experience, skills, and motivation of testers. Second, the actual importance of documenting test cases before executing the tests is unknown. Third, practitioners report good defect detection and efficiency results of testing approaches not based on a predesigned set of test cases.

While only a few studies have looked at industrial practice, they show that test cases are seldom rigorously used and documented in industrial settings. Instead, practitioners report that they find test cases difficult to design and often quite useless. In practice, it seems that test case selection and design is much left to the individual testers. A large amount of testing in industry is performed without applying actual testing techniques or, e.g., any formal test adequacy criteria. Reasons for this can be many, but it shows the importance of the less formal testing approaches; such as exploratory testing.

5.2.2 Exploratory Testing

Exploratory testing is an approach that does not rely on the documentation of test cases prior to test execution. This approach has been acknowledged in software testing books since the 1970's. However, usually authors have not presented concrete techniques or practices for performing exploratory testing; instead treating it as an ``ad hoc'' or error guessing method. While test case design techniques set the theoretical principles for testing, it is too straightforward to ignore all the factors that can affect testing activities during test execution work. Exploratory testing approach can be characterized by the following five properties (Itkonen and Rautiainen 2005):

1. **Tests are not defined in advance** as detailed test scripts or test cases. Instead, exploratory testing is exploration with a general mission without specific step-by-step instructions on how to accomplish the mission.
2. **Exploratory testing is guided by the results of previously performed tests** and the gained knowledge from them. An exploratory tester uses any available information on the target of testing, for example a requirements document, a user's manual, or even a marketing brochure.
3. **The focus in exploratory testing is on finding defects** by exploration, instead of systematically producing a comprehensive set of test cases for later use.
4. **Exploratory testing is simultaneous learning** of the system under test, **test design**, and **test execution**.

5. The **effectiveness** of the testing **relies on the tester's knowledge, skills, and experience**.

Some student experiments comparing exploratory and test case based testing approaches are already available. The results of these studies suggest that the exploratory approach is more efficient as it seems to require less effort, and usually there is no significant difference between the approaches with respect to defect detection effectiveness (Houdek et al. 2002; Itkonen et al. 2007). For more references to research on exploratory and experience based testing refer to doctoral dissertation by Juha Itkonen (Itkonen 2011).

The most commonly proposed benefit of ET is its ability to increase the effectiveness of testing in terms of number and importance of found defects. Some practitioners have even reported, based on their personal experience, that in some situations ET can be orders of magnitude more efficient than scripted testing. Exploratory testers can focus on the suspicious areas of the system based on the information on the actual behavior of the system, instead of only relying on the specifications and design documents when planning the tests. One important aspect behind the efficiency of ET is its ability to minimize preparation documentation before test execution. This is also an advantage in a situation where the requirements and the design of the system change rapidly or at the early stage of product development when some parts of the system are implemented, but the probability for major changes is still high.

The proposed benefits of exploratory testing include also many other things in addition to the efficiency. Examples of the benefits are simultaneous learning, ability to perform testing without comprehensive documentation, rapid feedback, the ability to utilize tester's creativity, and non-reliance on documentation.

Simultaneous learning in exploratory testing is based on the idea that when testers are not following pre-specified scripts, they are actively learning about the system under test and gaining new knowledge about the true behavior and the failures in the system. This is claimed to help testers come up with better and more powerful tests as testing proceeds.

As exploratory testing does not rely on comprehensive specifications or other documentation it is possible to apply also in a situation where the documentation does not exist or is poor. Exploratory testers can easily utilize all the experience and knowledge of the product gained from various other sources.

Exploratory testing can be considered an agile approach to software testing. In ET it is possible to achieve a rapid flow of feedback from testing to both developers and testers. This feedback loop is especially fast, because

exploratory testers can react quickly to changes to the product and provide test results back for the developers.

The exploratory approach lets the tester freely explore without being restricted by pre-designed test cases. The aspects that are proposed to make exploratory testing effective are the experience based knowledge, creativity, and personal skills of the tester. Common benefits of and Reasons for using exploratory testing are:

- Writing detailed test cases for everything is difficult and laborious
- Easy to test from the user's viewpoint
- Enables utilizing the testers' experience and creativity
- Rapid flow of feedback from testing
- Adapts well to changing situations
- More versatile and goes deeper into the tested features
- An Effective way of finding defects (minimized documentation before test execution)
- Even if test cases are used the defects are found by exploring
- Gives a good picture of the overall quality of the system in a short time
- Testing things that are hard to describe (e.g. look and feel of the system)
- Ability to perform testing without finished or comprehensive documentation
- A good way of learning about the system

There are also shortcomings in the exploratory testing approach. The most challenging shortcoming of exploratory testing is the difficulty of tracking the progress of individual testers and the testing work as a whole. It is hard to find out how work proceeds, e.g., the feature coverage of testing, because there is no planned low level structure that could be used for tracking the progress. Other challenges with ET are managing test coverage, evaluating the quality of the testing work, strong reliance on the performance of individual testers, and difficulties in repeating tests and reproducing identified defects.

5.3 Intelligent Manual Testing

The concept of exploratory testing relates to the ability to learn, design tests, execute tests, and interpret results as parallel activities of testing work. An important characteristic of exploratory testing is that tester continuously learns and designs new tests and improves the plan based on what one learns and observes about the behavior of the tested software. In exploratory testing these testing activities are not separated in time and the overhead of transferring knowledge between people between testing activities is avoided. This characteristic implicates that the test cases cannot be designed or documented prior to the actual testing (test execution) work.

Exploratory characteristic is one aspect of more general experience based approach to software testing. In the rest of this chapter, we discuss *Intelligent Manual Testing* (IMT), which is a wider concept, including exploratory testing, but not excluding all use of test cases or designing tests beforehand.

Intelligent Manual Testing is a concept that refers to all testing that is performed by human testers *and where some aspect of the testing is strongly dependent, or based, on the personal skills and knowledge of the tester during test execution work*. Thus IMT includes also testing that is not highly exploratory. IMT can be testing where the tester systematically uses documented test cases or designs tests before the actual test execution work. The important aspect in IMT is that testing is not purely based on detailed documentation; instead, personal knowledge and skills are utilized during execution time, not just in pre-design activity. In IMT, testing is recognized as professional work that can be supported by applicable documentation, and the tester is not treated as a robot that mechanically executes the tests.

The main properties of IMT can be summarized as three points:

- The tester is not treated as a test execution robot.
- Testing is strongly based on utilizing the tester's skills and knowledge during testing when tests are executed and defects revealed.
- The tester aims actively at utilizing past experiences and knowledge as well as new understanding learned during the testing activities.

In IMT, the tester takes care of important aspects of the testing during test execution without detailed pre-documented instructions. The aspects that in IMT are efficiently handled based on the testers' skills and experience can be, for example:

- Designing test cases (inputs and steps)
- Analyzing the expected results, correct behavior, and identifying the defects
- Creating relevant test data and realistic usage scenarios
- Identifying variations, exceptions, special cases, etc.
- Bringing in destructive attitude and negative testing

Any of these aspects can also be carefully and systematically pre-designed and documented in detail, if needed. However, if no freedom and room for experience, creativity, and exploring is left for the tester at the test execution time, the testing is not IMT any more. In IMT the tester may design and document tests beforehand, but the testing is still not based only on strict conformance to the documented test cases. Actually, in IMT the documented tests can also be the result of testing. Usually, the best way to find useful and efficient tests is to perform the testing and learn by doing. If

the same tests are needed to be repeated many times for some purpose, it may be good to document the tests on some, more or less detailed, level.

5.4 Heuristics for an Intelligent Tester

Heuristics are a good way of communicating general testing knowledge. Heuristics are "rules of thumb, educated guesses, intuitive judgments or simply common sense. Heuristics provide informal advice for solving a problem that in most cases is helpful, but is not guaranteed to be true or applicable in every situation. Since low level testing practices are highly situational and strongly dependent of the context, e.g., technology, application domain, type and criticality of the software, it is hard to give absolute rules or form general theories of testing practices. In such a context, heuristics are applicable because they are meant to be applied, not taken as strict rules.

In this section seven heuristics for an intelligent tester are presented. These heuristics are based on our observations when we have studied real testing work. The context of these heuristics is test execution work, i.e. the actual testing and defect finding in front of the tested software. More testing heuristics can be found e.g. in the work of Kaner, Bach and Pettichord (Kaner at al. 2002) and Bolton (Bolton 2005).

5.4.1 **Remember to Explore**

Exploring is an important part of software testing. Remember always to explore the tested functionality and features of the system even though you were using predefined test cases. Test documentation can never be so exhaustive as not to leave any room for exploratory search for the difficult to detect defects.

- Explore freely around the test case.
- Explore to cover the details more deeply.
- Explore to better understand the next thing the real user would do when working with the system.
- Explore to cover exceptional and abnormal scenarios.
- Explore all the time besides your actual tested features to perform continuous smoke and regression testing.
- Explore to learn and improve your testing.

5.4.2 **Evaluate the Outcome Carefully**

The results of a test (case) are not always obvious. The results are usually not comprehensively documented, either. An important part of the professional skills of a tester is the ability to understand and check all the

effects that the results of a test should cover. Try to understand all consequences and effects that the tested functionality should or could have in the system and check them.

- Check things that should change
- Check things that should not change
- Check things that might have changed
- Consistency helps to reveal incorrect behaviour --- is the outcome consistent with the expectations, requirements, other features, purpose of the function, past behaviour, etc.

5.4.3 Investigate Interactions

Difficult defects are not revealed when you focus just to one feature or one variable at the time. Difficult defects are related to combinations of more than one feature or affecting variable. Focus on feature interactions. How do features affect each other?

- What features can be used in combination?
- What features use the results of this feature?
- What features affect the data this feature depends on?
- What features do similar things than this feature?
- If this feature fails, where would it be visible?
- If some other feature fails, how does it affect this feature?

5.4.4 Variation is Good

In testing, it is good to do things in different ways. Repeatability is required for regression testing and measuring certain qualities, e.g. performance. However, when the goal is revealing new defects, varying and diverse tests are always better than static ones.

- Use different inputs
- Use different steps
- Use different data
- Use different order
- Use different sequence and timing
- Do tests in different states and modes of the system

5.4.5 Focus Your Testing

The amount of possible tests is unlimited. Focusing the testing efforts to essential areas is crucial for success. The testing must be focused on areas that are most likely to fail, are most risky, or are most important for the customer or user.

- Focus on known risks
- Focus on limitations and restrictions
- Focus on exceptional and unexpected scenarios

5.4.6 Remember the Purpose

Tester represents the customer and the user. The user's viewpoint should always be kept in mind in testing and the tester should see the findings also through the end user's eyes. Simulating the real usage of the system is an important part of testing that reveals important omissions, problems and shortcomings that are not apparent if features are analyzed from technical viewpoint in isolation.

- Test realistic scenarios
- Perform the most important tasks that the software is designed for
- Perform end user’s tasks entirely, including realistic size, duration and complexity.

5.4.7 Keep an Eye on Coverage

Even though freestyle exploratory testing seems to be an efficient way of revealing defects, it is still important to be able to plan and track the testing efforts. You should somehow structure your testing to have at least a high level plan to follow and to be able to say what you are going to do. You need to be able to track and show how the work proceeds and, after the work, tell what you have done. Testing involves, in most cases, a great number of combinations of features, environments, and other details. These combinations cannot be managed without a systematic approach.

- Plan what you will cover
- Know what you have covered
- Cover the important combinations

5.5 Practices of an Intelligent Tester

In our research, we have identified 23 practices of an intelligent tester. The practices were found by observing the actual work of testers in four different companies. Based on the observation data we were able to identify and describe a large amount of practices that the testers used, intentionally or subconsciously, during the test execution. All observed testers were IMT testers according to the above definition of intelligent manual testing. The practices presented in this section are the real means that the testers used during test execution work to guide their testing activities and reveal the defects efficiently. The focus is purely on the actual testing work and how the testers' knowledge and experience are utilized for detecting defects. No separate test planning practices or techniques for test case pre-design are presented.

The IMT practices can be classified in five different categories according to the main idea that the practices rely to and the abstraction level the

practices are applied. The next sections discuss these practices. For detailed descriptions of the identified practices, see Appendix B.

Table 17. Classification of the IMT practices

Test session strategies	Exploratory
	Documentation based
Testing techniques	Exploratory
	Comparison
	Input

The test session strategies are high level practices that the subjects used to give an overall structure to their testing work. Session strategies gave the general guideline of how to proceed in testing and what aspects to cover. In combination with session strategies, subjects typically used free exploratory testing or some test execution technique (see Table 17) to test the details of each feature. The session strategies are further divided into subclasses of exploratory strategies and documentation based strategies.

The testing techniques are practices that the subjects used for testing individual or tightly related features, or such small details as input values. We categorized the techniques into three subclasses: Exploratory, comparison and input techniques.

5.5.1 Exploratory Session Strategies

The exploratory strategies were used to guide and give structure to exploratory testing. An example of such strategy is *User interface exploring* where a tester structured the testing through user interface features and proceeds from feature to feature covering all UI features, but relied on experience-based approach in testing each individual feature. Another example is *Exploring weak areas* that relied on the tester's experience and personal knowledge of potential weak areas, not based on documented analysis. Rest of the strategies are *Aspect oriented testing, Top-down functional exploring, Simulating a real usage scenario, and Smoke testing by intuition and experience.*

5.5.2 Documentation Based Session Strategies

The documentation based session strategies, in contrast to exploratory strategies, are used to guide experience-based testing using documented tests or other documentation. The used documentation can be test cases that are described on varying level, or release notes and defect reports. The documentation is used as a checklist to give structure for test execution or as high-level test cases that are extended and deepened by experience-

based approach. An example of such strategy is *Data as test cases* where test data was documented and used to give structure for testing and manage coverage. Instead of functional steps the testing was guided by situations defined in terms of test data. The strategy was used in testing a financial system where the test data represented different kinds of customers with different properties, services, and personal situations. This data set was used for covering a representative set of situations in experience-based testing of the features of the system. Other strategies are *Exploring high-level test cases, and Checking new and changed features*.

5.5.3 Exploratory Testing Techniques

The exploratory techniques are techniques that are used for exploring one isolated functionality or a single function. Most of the exploratory techniques are based on hypothesis of a certain type of defects that the technique aims to reveal, or a certain typical situation where defects are often revealed. For example, *Simulating abnormal and extreme situations* is a technique for testing extreme situations or scenarios. The aim is to evaluate how a function performs on and beyond its limits and reveal problems of handling abnormal and stress situations where the limits or rules of normal operation are violated. Other techniques are *Testing alternative ways, Exploring against old functionality, Persistence testing, Feature interaction testing, and Defect based exploring*.

5.5.4 Comparison Techniques

The comparison techniques are used for evaluating the test outcomes and making difference between correct, expected; and incorrect, erroneous behaviour during testing, i.e., how to recognize a defect. Subjects needed comparison techniques to analyse complicated features and evaluate the expected outcome especially when the correct function was not unambiguously specified or involved many aspects that must be taken into consideration. For example, *Comparing within the software* is a technique for comparing similar features in different places of the same system. The aim is to assess if a feature works correctly or not by investigating the consistency of functionality inside a software. Other techniques *are Comparing with another application or version, Checking all the effects, and End-to-end data check*.

5.5.5 Input Techniques

The input techniques are used for detailed testing of individual input values or set of related inputs. Input techniques are used to manage covering the details of a feature and selecting relevant test cases or values for testing. For example, *Testing boundaries and restrictions* is a technique that focuses on testing the boundary values and other restrictions of input data. The aim is to cover all explicit and implicit restrictions of a single function and reveal defects that are associated with handling boundaries and restrictions. Other techniques are *Testing input alternatives and Covering input combinations*.

5.6 Utilizing the IMT Practices and Heuristics

In previous sections, we presented the concept of Intelligent Manual Testing, seven heuristics for an intelligent tester and 22 practices of an intelligent tester. We are now ready to outline guidelines for utilizing these practices in your own organization. There are several ways you can utilize these ideas. The obvious use is for training testers to better understand the different aspects of software testing and teach them practical means of doing good testing work. Other uses of these practices are using the practices to guide testing and help plan and document the tests efficiently. The practices can also help tracking the progress of testing work and manage test coverage. In the following, we describe different ways of utilizing the IMT heuristics and practices.

5.6.1 Using IMT Practices for Training Testers

Make novice testers experienced fast – by training IMT Practices

The IMT heuristics as well as the IMT practices capture a lot of good, practical software testing knowledge. This knowledge is different in nature than typical theoretical text book content. This knowledge captures small pieces of the experience of a large pool of real people performing software testing. Using this knowledge for training you can transfer such knowledge that testers have gained by experience when doing real testing work. We have seen that the basic software testing techniques as they are described in books and papers are hard to apply in practice and the true testing knowledge is gained in practice. Training IMT heuristics and practices to testers is the first step towards a faster way of acquiring the experience based testing knowledge.

The IMT practices provide names and descriptions for experience based test ideas. This helps testers communicate and apply those ideas in practice. By learning these practices the testers have a wide set of explicit means for doing their work in practice. By using the basic classification of the practices and the descriptions the testers are able to choose practices for their specific situation and needs. The practices work also for better and more efficient communication among testers.

5.6.2 Using IMT Practices for Guiding Testig Work

Tell the tester what to do – not how to do it on detailed level

Most of the IMT practices are focused on the low level details of executing the tests and defect detection strategies. This is the level at which traditional test case descriptions are documented. The IMT Practices can be used for guiding the tester instead of detailed test cases. The traditional test case design techniques, such as equivalence partitioning or domain testing, are based on theories of what kinds of defects there typically exist in software and how the defects can be efficiently revealed. The same idea is implemented by the defect detection strategy based IMT Practices, marked with the bomb symbol in (see Appendix A). The difference is that in IMT you assume that the tester knows how to do the testing based on the instructions that identify what should be tested using a specified IMT Practice. Similarly, an important part of the traditional test cases is detailed description of the expected results. In practice, however, it is many times safe to leave the analysis and recognizing the defects for the tester. Also for this you can find applicable IMT Practices, marked with the scale symbol (see Appendix A). In addition, by applying an IMT Practice the tester is actively analysing the tested feature, not just passively following a script. This way the tester can better maintain focus and pay attention to the right issues. The tester has a better idea what he is doing and what is the goal and purpose of the tests.

5.6.3 Using IMT Practices for Test Design and Documentation

Make test design and documentation more efficient – by using IMT Practices

Using IMT practices has potential for saving a lot of test documentation work, because the practices describe a general strategy, or method, in such form that an intelligent tester is capable of applying it without detailed step-by-step instructions.

The practices can also be applied in test planning for describing the testing approach. The practices describe higher level test ideas and ways of covering the functionality. Using these practices the testing approach and high level test ideas can be described in easier and more efficient manner.

Practices can also be used to create organization specific tailored practices for a specific context, e.g. "web application testing in financial domain". Tailored practices of this kind could work directly as reusable components for creating test design specifications. Don't use test cases to teach testers how to test, teach them IMT practices instead and use test documentation for planning, managing, and tracking the testing work.

5.6.4 **Use Session-Based Testing to Manage IMT**

One good process for managing exploratory testing or IMT in general is Session-Based Testing (SBT). In SBT the testing work is not managed as test cases, instead, the work is divided into time-boxed *sessions* and the work is planned and tracked using these sessions as the basic unit of testing work. The process works by planning the testing tasks as fixed-length (e.g. 2 hour) sessions. The testing work to be done in each session is described as high level guidance called a charter. When a tester performs one test session he produces a log which is reviewed face-to-face with a test lead, who is responsible for tracking and managing the sessions, creating new sessions and collecting and tracking the results. The main benefits of SBT are following:

- Enables planning and tracking Intelligent Manual Testing
    - Without detailed test (case) designs
- Uses time-boxed testing sessions as the planning and tracking mechanism instead of test cases and suites
    - Planning and tracking testing work in small chunks
    - Dividing testing work in smaller tasks
    - Tracking testing work in time-boxed sessions
    - Can help getting testing done when resources are scarce
- Efficient – no unnecessary documentation
    - Utilizes tester's skills and knowledge directly during testing
- Agile – testing is easy to focus on most important areas based on the test results and other information
    - Changes in requirements, increasing understanding, revealed problems, etc.

SBT matches nicely with IMT Practices. The practices can be used to describe the details of the test charters and simplify the logging. Also the practices for managing test coverage suit well to complement SBT. For more details of how Session-Based Testing works see the following articles:

- Session-Based Test Management by Jonathan Bach
    - http://www.satisfice.com/articles/sbtm.pdf
- Adventures in Session-Based Testing by James Lyndsay

- http://www.workroom-productions.com/papers/AiSBTv1.2.pdf

5.6.5 Managing Test Coverage in IMT

Managing the test coverage is one of the challenges of exploratory testing approaches. Typically test coverage is managed in terms of source code level coverage criteria or how well test cases cover the requirements or functionality. However, the IMT Practices provide a lot of help also for tackling this challenge. Many of the IMT Practices actually aim at managing coverage of some aspect of the tested software. By using these practices, marked with the checkmark symbol, you can plan and track the coverage of IMT from various viewpoints. This way managing test coverage should be possible without detailed test case description.

Example of how to manage test coverage from multiple viewpoints using IMT Practices: Use *Checklist Exploring* practice to cover all functionality documented in the functional specification. For each covered function use *Covering all input alternatives* and *Exercising boundaries and restrictions* practices to ensure good enough coverage of the details of each function. In addition, use *User interface exploring* practice to ensure that all GUI level functionality is covered even if it's not described in the functional specification.

References

Bolton, M., 2005. Testing Without a Map. *Better Software Magazine*, January 2005.

Houdek, F., T. Schwinn, and D. Ernst, "Defect Detection for Executable Specifications - An Experiment", IJSEKE, vol. 12(6), pp. 637-655, 2002.

Itkonen, J. and K. Rautiainen. "Exploratory Testing: A Multiple Case Study", in Proceedings of the International Symposium on Empirical Software Engineering (ISESE 2005), pp. 84-93, 2005.

Itkonen, J., M. V. Mäntylä and C. Lassenius. "Defect Detection Efficiency: Test Case Based vs. Exploratory Testing", in proceedings of the International Symposium on Empirical Software Engineering and Measurement, 2007.

Itkonen J., *Empirical Studies on Exploratory Software Testing*, Doctoral dissertation, Aalto University School of Science, Novembr 2011.

Juristo, N., A. M. Moreno, and S. Vegas, "Reviewing 25 years of Testing Technique Experiments", Empirical Software Engineering, vol. 9(1-2), pp. 7-44, 2004.

Kaner, C., Bach, J., Pettichord, B. *Lessons Learned in Software Testing*. John Wiley & Sons, Inc, New York, 2002.

6 Pair Programming

6.1 Introduction

In *pair programming* (PP) two persons design, code and test software together at one computer. The *driver* controls the keyboard. The *navigator* observes the driver's work trying to find defects, but also thinks at a more strategic level. The persons communicate actively and switch roles periodically. (Williams 2002)

To some degree software has been done written collaboratively for decades, and the first two experiments studying such a practice were reported already in the 1990s (Wilson 1993, Nosek 1998). However, only after year 2000 PP started to become well-known as an explicit practice, because it was chosen as one of the mandatory practices in the popular Extreme Programming (XP) software development approach (Beck 2000). XP requires using PP for all production code, but in other contexts there is no reason why it could not be used in a less disciplined way. According to many surveys conducted in recent years, PP has been taken into use in industry at least to some degree (see e.g., Salo and Abrahamsson, 2008; Schindler, 2008; Begel and Nagappan, 2008).

The interest in PP is natural because PP has been proposed to have a positive effect on, e.g., quality of code and design, knowledge transfer, learning, team work, and work satisfaction. The negative effects may include increased development effort, increased mental exhaustiveness of work and the fact that not every developer likes doing PP. The reasoning for adopting PP is that the expected benefits such as improved quality and learning increase the overall productivity in the long run even though initially finishing a task using PP may require more effort than when working alone.

The current knowledge of PP is rather inconclusive what comes to its effects and to the details of how it should be applied in different contexts. This can be seen in our summary of literature in chapter 2. The further chapters discuss our own experiences of three cases, where we studied PP.

These cases ranged from a rather disciplined PP use in moderately sized student projects to quite informal use of PP in large-scale software product development in industry.

6.2 Effects of pair programming

6.2.1 Context dependency of the effects

The effects of PP have been studied mostly with students who have made small, isolated tasks or small projects. Experiences of applying PP for real in the industry have been reported in several papers, but they are typically experience reports where rigorous data collection has not been carried out.

The existence and magnitude of the effects of PP may be influenced by many context factors such as characteristics of the developers including their familiarity with PP and their partners, roles, communication, partner switching, type of work, development process and tools, and workspace facilities (Gallis 2003). The research on the effects of PP is still inconclusive and one of the main reasons for that may be the context dependency of the proposed effects of PP (Hannay 2009).

Organizations adopting PP often have the possibility to control many of the context factors, but they seem to understand poorly what would be a good context for PP. The difficulty can be seen in the practical questions we have faced when observing the adoption of PP in non-XP, industrial contexts:

What kind of work is done using PP?
- considering different activities (analysis, design, code, test) and difficulty of work

Who proposes the use of PP for a task and when?
- managers, developers or both?

Who pairs with whom considering e.g. competencies, experience and personalities?
How long does the same pair work together?
If a pair does not do a whole task together:
- How much do they work together?
- Do they work separately with the same task?
- How do they synchronize after separation?
- How do they communicate during separation?

What kind of infrastructure is good for PP?
How does one behave during a PP session?
How often are the driver/navigator roles switched?

XP gives some extreme answers, e.g., everyone should use PP for all development tasks from the start to the end. However, it seems than even XP projects seldom apply such an extreme approach. Generally, the answers are likely to depend on the goals set for the use of PP, e.g. ensuring

high quality vs. mentoring a novice, and on the fixed context variables, e.g. the characteristics of the developers. Different answers lead to different application of PP.

6.2.2 Proposed Effects

Table 18 summarizes the proposed effects of PP classified under learning, quality, effort and schedule, and human factors. Most of the effects are positive, and the negative ones include potential increase in effort, and higher exhaustiveness of work. The studies are still inconclusive, and many of the effects have been poorly studied especially in the industry. Thus the summary should be taken as a long list of potential effects rather than as a list of proven effects.

Table 18 Proposed effects of pair programming

Area	Effect
Learning	increased learning of work related topics smaller amount of critical knowledge known by only one person
Quality	smaller number of defects better solutions to problems higher comprehensibility of code and design
Effort and schedule	increased total effort for a task more accurate effort estimates finishing tasks faster better schedule adherence
Human factors	higher satisfaction and confidence in work higher exhaustiveness of work improved trust and teamwork more discipline more courage to attack difficult things, such as refactoring, and to admit ignorance

Even if PP required somewhat more effort, the long term benefits may compensate this cost.

6.3 Case Eddy

6.3.1 Case description

The Eddy project was carried out in a large telecommunications company in Finland. The project goals were 1) to develop an internal reporting system for the company and 2) to pilot agile practices.

The project team included four developers, who had not earlier worked with each other. Their positive attitude to using agile practices was ensured before recruiting them. Three developers had not used PP before the project and one had used it for about a month. Initially the developers' attitudes to PP varied from slightly negative to quite positive. They had 4–10 years of programming experience of which 1.5–4.0 years with the Java technologies used in the project.

The team created their development process by picking up practices from several agile methodologies. A person from the same organization acted as a coach, i.e. helped the team in issues related to the work practices.

The developers shared the same room and had visual contact with each other. There were no private workstations available. The coach and a person acting in the customer role had their own rooms close to the team.

6.3.2 **Application of pair programming**

Adoption

During a pre-project iteration the team was given lectures about agile development covering also PP, and each developer spent eight hours using PP. The developers considered this was sufficient to start doing PP efficiently.

PP was adopted thoroughly from the start of the project. The team had only two high-end workstations and two low-end workstations. This practically forced them to use a lot of PP. The developers did not criticize the setting, probably because of their approval of experimentation with agile practices, but also because PP was soon accepted as a good practice. Afterwards, all the developers considered using PP easy both absolutely and compared to the use of some other practices such as test-driven development.

The degree of pair work in each iteration is shown in Table 19. 72% of the programming effort in the project and 52% of all effort was done in pairs. The use of PP was slightly lower in iterations I2, I5 and I6 because refactoring and fine-tuning activities took time away from developing new features. These activities were considered easier and therefore PP was used less. The team size also decreased to only two developers after I4. Iterations I1-I4 reflect the normal state of the project better and for them the amounts of pair work were 77% of the programming effort and 66% of all effort.

Table 19 Proportion of PP

	Pre	**I1**	**I2**	**I3**	**I4**	**I5**	**I6**	**Σ**
Persons	4	4	4	4	4	2	2	
Days	4	9	10	10	11	10	17	71
All effort	129h	233h	235h	264h	274h	147h	222h	1505h
of which pair effort	N/A	135h	117h	159h	140h	42h	37h	628h
	N/A	82%	58%	73%	57%	28%	16%	52%
Programming effort	32h	100h	127h	138h	107h	18h	60h	582h
of which pair effort	32h	91h	79h	109h	85h	12h	11h	419h
	100%	91%	63%	79%	79%	67%	18%	72%

Pair formation

Pairs were formed in daily meetings. First the formation of pairs was affected by e.g. trying to avoid pairing the two most experienced developers, and also by the frequent habit of smoking by two of the developers. The same two people continued to pair the next day if the same task continued. After I1 the team considered that such an infrequent rotation was not sufficient for good knowledge transfer. Frequent rotation was emphasized more in I2, but it still did not happen very often. Therefore, starting in I3 the team formed the pairs by casting a lot each morning so that the pairs always rotated compared to the previous day. If a pair did not finish their task by the end of the previous day, one of them continued it with a new partner. Regular rotation worked well and simplified pair formation, because in a four-person team rotating pairs after a task ends requires waiting until the other pair also completes their task.

Pair programming sessions

Each task was usually self-assigned to a pair who worked together with it. If a pair separated the partners either continued with the task separately or sometimes one of them could start another task.

When starting a new task the pair made some specification work and consulted the customer for the details of the task. Then the pair did some design work and started programming. Sometimes the opinions on the amount of design required before coding differed and this could cause disagreement between the partners.

Communication was continuous with no silent moments. If a person was already familiar with the problem he acted as the driver and explained continuously what he did. The developers felt it impossible to act as the navigator without knowing what the code did.

PP could last the whole day, interrupted only by lunch or other breaks every now and then. The roles were switched 2–3 times a day, typically

after the lunch break or when the driver took a personal break and temporarily left the workstation.

The use of PP got looser later in the project. For example, the developers did not necessarily sit at the same workstation all the time when programming. Especially when very simple things were programmed the partner sometimes went to do other work or took a break.

Targets for pair programming

Developers considered PP especially useful for complex tasks. When doing simple copy and paste coding the navigator soon lost his interest in the work. Some straightforward tasks were done alone if they were easy to split into two parts where the other part could be performed at the low-end workstation by the partner.

6.3.3 **Lessons learned**

Learning PP took place quickly and developers considered its use very easy. Having a smaller number of high-end workstations than there were developers certainly contributed to the quick adoption and rigorous use of PP. Surprisingly, the developers did not criticize the lack of own computers.

It seems that PP was better suited for difficult tasks. The developers avoided its use for trivial tasks if it was possible to split the task in two parts, i.e. one for each partner. The development effort was considered lower for PP than for solo programming with complex tasks but for easy tasks the situation was reverse.

Initially pair rotation occurred only after a task was finished. In order to improve knowledge transfer the team started to actively rotate pairs each morning even if their tasks were not finished. This change seemed to increase the knowledge transfer within the team. However, there was a drop in productivity after the change, but the productivity rose again in the next iteration.

The driver/navigator roles were switched only 2-3 times a day. This has been said to passivize the navigator, but in this case the partners maintained active communication. The use of PP became slightly more relaxed later in the project and the pairs could split up when the driver did some easy programming.

There were many dependencies between PP and other practices. PP and especially the presence of the navigator increased discipline in using many other practices, such as test-driven development, coding standard and frequent integration. PP increased collective code ownership because at least two persons participated in each task. Two persons were also better

able to solve problems related to writing testable code and designing good unit tests. Collaborative task ownership allowed the pair to jointly celebrate the achievements, and ensured that both partners were willing to participate in completing the task. Test-driven development increased communication of ideas between the partners, and daily meetings helped forming the pairs.

Developers considered that PP contributed to the low defect counts in the developed system. However, contrary to what has been proposed, the navigator seldom found any defects during the programming work, meaning that some other aspect of PP, such as mere presence of a partner, or designing and test-driven development together, helped to avoid injecting defects.

All developers considered that PP increased their knowledge of the system more than solo programming. Two developers considered that PP helped them learn the development tools better, but the other two found no difference in this knowledge transfer aspect. The developers ranked PP as the most important practice for increasing team communication.

The team spirit was very high at least partially thanks to PP. Nobody was against PP and half of the developers liked it even more than solo programming.

6.4 Case Proco

6.4.1 **Case description**

The case organization was a department in a medium-sized Finnish software product company. There were a few dozens of developers divided into several development teams, each having a senior developer as a team leader. All developers were sitting in the same open office containing many cubicles and desks. The department was responsible for the development of a large and very successful software product, with a development history of 15 years. The development languages were C/C++, and the most widely used development environment was Visual Studio.

6.4.2 **Application of pair programming**

Adoption

The motivation for adopting PP was to improve software quality and increase knowledge transfer among developers. The case organization did not aim for an XP style full scale use of PP, but for using it only when it would be most beneficial. Before the adoption, the developers' experience of

PP varied a lot due to PP experiences in their previous work and studies. Most developers had a few dozens of hours PP experience, a few had no or almost no PP experience, and a few had used it for hundreds of hours.

A team leader became the PP champion, who took the responsibility for the department's PP guidelines and for promoting the use of PP in the whole department. He created and introduced the initial PP guidelines that are summarized below:

- PP should be used in particular when new developers join a team, when knowledge needs sharing, or when doing difficult or error-prone tasks.
- Teams should select the features that would benefit most from the use of PP. PP should be used in particular for the specification and design activities for these features. A pair should work together for 10–100% of a feature's development effort. Solo development should not be done more than 10h between two PP sessions.
- Each feature still has only one responsible person.
- Developers select their partners themselves.
- PP sessions should last 0.5–3h, and be allocated in advance.

Initially the developers had quite neutral attitudes to PP. After using PP, the attitudes improved, and after two years of PP use were as good as the attitudes to solo programming.

The majority of developers continuously reported wanting to use more PP, but inadequate attention to resourcing PP, disturbing noise generated by PP, and inconvenient work spaces for PP were hindering an increase in its use. About a year after the initial adoption, a dedicated PP room was instituted. It provided both the needed space and the technical infrastructure to support PP, and helped shield other developers from the noise created by the pair programmers.

About two years after the initial adoption of PP, it was used for ca. 15% of development work. At the same time the desired proportion was 25% of development work on the average.

Pair formation

Pair formation worked pretty well, but there were some contradictory opinions on the best ways to do it. Several developers considered the developers should have the final decision on the use of PP and on the partner. However, some other developers recommended planning the use of PP in weekly team meetings, and team leaders should be more active in ensuring that PP gets done. The latter opinions may be due to problems of getting more senior developers to do PP with the juniors even if the juniors would like to use PP. The opinions on good partner combinations were quite similar amongst the developers. There should be a senior and a junior

developer, or the partners should have knowledge areas that complement each other for the task at hand.

Pair programming sessions

Typically, the partners agreed in advance on arranging a PP session and reserved a couple of hours for it. Most developers considered 1.5–4 hours a suitable session length. Shorter sessions were considered inefficient, because it takes a while to get to speed, and in longer sessions, concentration decreases due to the exhaustiveness of PP. A developer mentioned the lack of breaks as a reason for the exhaustiveness. It may be that when working intensively with a pair you actually need more breaks than when working alone, but in practice you end up taking fewer. Between the sessions, the person responsible for the task typically continued on it alone.

Only about 20% of the developers switched roles during the PP sessions. They considered it a good practice without reporting any drawbacks. Half of the developers switched roles only when starting a new session and the rest switched even less frequently. However, even in the sessions where the roles were not switched both partners were very active. All the developers did not have a similar development environment, e.g. the same text editor, which was one reason for the reluctance to switching roles.

Targets for pair programming

In 75% of the cases, the reason for using PP was getting more information on a possibly difficult task. Other reasons were improving quality and finishing faster. The developers considered PP especially good when tutoring new developers. Also all kinds of difficult tasks were good targets for PP. Software specification, challenging coding and problem solving were commonly mentioned examples. Using PP in the beginning of a task to learn more about it from a more experienced developer was considered useful. Poor PP situations were routine coding and when neither of the partners was well prepared or knew enough about the task.

Problems and improvements in the organization of pair programming

The problems related to organizing for PP included being too busy to use PP related to the other developers' tasks, difficulties in finding common time, lack of encouragement from team leaders, and not considering PP in project planning. Some developers had not used PP because their tasks had not been suitable for PP, i.e. their tasks have been simple and/or independent.

About a year after the initial PP adoption, the PP guidelines were updated in order to create a more encouraging and more positive atmosphere for PP. The updated guidelines explicitly mentioned that both the team leader and any developer may propose the use of PP. The proposals about the features for which PP would be used, should preferably be made when a four-month long iteration's work is planned and assigned to developers. The proposal should identify the partners, and roughly how much they should use PP for the different activities related to the development of the feature, e.g. specification and coding. In addition, it was emphasized that team leaders should encourage using the planned amount of PP for the selected features.

Problems and improvements in the infrastructure of pair programming

There were several problems related to the infrastructure for PP. The developers' own work spaces were quite cramped for doing PP. The noise from PP also disturbed others in the open office. Therefore, PP was sometimes done in a meeting room using a developer's laptop. Not all developers were used to same development tools, which made role switching impossible in these cases.

After about a year of PP use, a PP room was adopted. It contained a desk with *two* computers with large displays, a long, straight table, rolling chairs, and a whiteboard. The developers worked over a remote connection to their own desktop computer in order to have a familiar development environment. The room could be reserved using the company's meeting room reservation system.

Separate rooms for PP may be needed to solve the noise and space problems related to using PP in a typical open office.

The PP room became the preferred place to do PP by 89% of the developers. The simplicity of doing ad-hoc PP sessions, and being closer to the others if help was needed, were the only benefits of the doing PP in the own workspace. The benefits of the PP room included avoiding disturbing others and being disturbed by others, and having the required infrastructure including enough space, two chairs and two workstations immediately available.

Usually both partners worked at the same computer, but the additional computer allowed more efficient work when some things needed to be

clarified e.g. by finding information, or when trivial code was written and the partner could do something else during that.

6.4.3 **Lessons learned**

A set of simple PP guidelines was created and communicated to all developers in order to spread good experiences on how to use PP. After having more experience on PP, the guidelines were revised to tackle, e.g. problems with lacking resource allocation for PP.

On the average the usage of PP increased continuously but slowly. Since the use of PP was voluntary, its use varied among developers. Two years after the initial adoption, the majority of the developers still wanted to use it even more than they were using it then.

Encouraging the use of PP instead of enforcing it probably helped lower the barrier to trying PP and created a positive atmosphere for PP. The positive atmosphere was seen in the developers' generally positive attitudes to PP. However, there were individual preferences of solo programming (SP) vs. PP and acting as the less vs. more skillful partner, which should be taken into account when forming pairs. However, a general expectation that junior developers would be more eager pair programmers than senior programmers did not get support in this case. There was no significant difference in preferences for PP vs. SP between senior and junior employees.

Initially, developers complained about inappropriate infrastructure for PP. This problem was alleviated by instituting a dedicated PP room that provided both the needed space and technical infrastructure to support PP, as well as helped shield other developers from the noise created by the pair programmers.

The perceived effects of PP were positive on learning, quality, schedule adherence of tasks, getting to know other developers, team spirit, enjoyment of work, and discipline in following work practices. It is interesting that the effect on enjoyment of work was positive, even though the perceived exhaustiveness of work was higher for PP. The perceived effect on task effort indicated a somewhat higher effort when using PP as opposed to solo programming. However, other benefits of PP might compensate for this, making the additional task effort acceptable from the perspective of overall productivity.

Voluntary, moderate usage of PP may increase developers' enjoyment of work even though using PP is a more exhaustive way of working.

6.5 Case Casino

6.5.1 Case description

We made an experiment where five equally experienced student teams of four developers conducted a similar project using the same development process, work practices, tools, and specifications. Three randomly chosen teams used pair programming (PP) and two solo programming (SP). The PP teams had to use PP for all development work whereas the SP teams were not allowed to use PP for more than occasional collaboration, i.e. not for implementing whole use cases together.

The experiment consisted of a nine-week project. The project included developing, testing and delivering a distributed, multi-player casino system using the J2EE technologies that were taught before the project (15h of lectures). The requirements for the system were described in a requirements specification containing, e.g., use case descriptions and HTML layouts for the web user interface. The teams were given a 10-page technical specification and a core architecture implementation including examples of suitable J2EE design patterns and a build script. The development tools used were Eclipse 3, J2EE 1.4 SDK, XDoclet, JBOSS with Tomcat, Hypersonic SQL, CVS and Ant.

The project effort was fixed to 400 hours, i.e., 100h per person. Everyone had to spend at least 75% of the effort in co-located team sessions lasting 4-8 hours. The project consisted of a one-week project planning phase followed by two four-week implementation iterations. The teams had to follow work practices such as iteration planning, collective ownership, version control, coding standard, continuous refactoring, unit testing, system testing, time reporting, defect reporting, and documenting. The prioritized project goals were to: 1) follow the defined work practices, 2) minimize the amount of defects, 3) implement as many use cases as possible, and 4) avoid wasting effort on activities that do not directly contribute to the project.

6.5.2 Application of pair programming

The PP teams had to use PP for all development work. PP was taught to the PP teams on a one-hour lecture. Pair rotation and role switching were proposed as good practices when introducing PP, but their realization was not measured. Length of the PP sessions were not instructed nor measured in any way.

6.5.3 Lessons learned

The PP teams had 29% lower project productivity than the SP teams. However, the reason was the considerably larger effort they spent for the first three or four use cases. The inefficiency was probably caused by the learning time involved in getting familiar with new PP partners and with the pair programming practice. After a learning time of a few dozens of hours, the PP teams spent 5% less effort than the SP teams for implementing the latter use cases. If the inefficient learning time is not taken into account, the productivity of the PP teams seems to be equal to that of the SP teams. In a typical software development organization the learning time can usually be neglected because most people already know each other and at least after the first pair programming project are familiar with pair programming. Even if there were still some learning time involved, the cost of a day or two per developer for learning is insignificant. The claim that pair programming is most useful with complex tasks was not supported by this experiment, at least from the perspective of the required effort.

Allow people to learn to use PP and get to know their PP partners before starting to evaluate the effects of PP.

The code written by pair programmers contained 8% less defects per use case when the responsible developers considered the code ready. However, the SP teams were much more successful in finding and fixing the defects, and in the end of the project they delivered systems with a lower number of defects per use case. This indicates that pair programmers write code with fewer defects, but this benefit may be lost unless careful system testing is performed.

The PP teams had slightly better design quality based on the method size and complexity metrics. However, the reason may be the potential

correlation between software size and these code metrics. The PP teams delivered systems with less functionality, and therefore these metrics may show better values for them.

In the PP teams developers generally had high involvement in more source code packages than in the SP teams. Probably related to this, there were generally more developers (1.8 vs. 1.4) in the PP teams with good understanding of each package, and each developer understood more packages (4.5 vs. 3.4) well. This indicates better knowledge transfer within the PP teams.

Even though about half of the developers in the PP teams enjoyed solo programming more than pair programming (and about half vice versa) most developers still liked working in the PP teams. Thus developers' feelings toward pair programming should not hinder its deployment. However, one of the teams was removed from the analysis because they abandoned the use of pair programming against the rules of the experiment, which suggests that they were strongly against pair programming after a couple of weeks of experimenting with it. The reason may be that PP really was an unsuitable practice for this team, but another reason, backed up by the fact that their productivity did not improve later, may be that the frustration about their slow progress led them to consider PP as a new practice as the main cause for their problems.

It seems that the use of pair programming leads to fewer defects in code after coding and better knowledge transfer within the development team without requiring additional effort if the learning time can be avoided. These benefits are likely to decrease the further development costs of the system and increase an organization's productivity due to improved competence of the developers.

6.6 Summary

In this section we summarize what we learned about PP in the three cases described above and based on the previous literature.

It may take a while to learn to do PP efficiently. In case Casino, we were able to measure this and found that the productivity rose until they had used a few dozens of hours of PP. In the Eddy project developers thought that they learned to do PP after only one full day of using it. One of the developers had already used PP, which may have helped also the rest of the team learn to do PP.

In cases Proco and Eddy the developers considered PP best for difficult tasks, and preferred to avoid it for routine coding. However, in case Casino we did not found correlation between the difficulty of a tasks and PP's effect

to task's implementation effort. It may be that the developers consider more the benefits of PP's, e.g. on quality and knowledge transfer, than its cost when evaluating its suitability. Also in case Casino there were no trivial tasks, i.e. all tasks contained at least some difficulty for the developers, who were learning the new J2EE technology.

In cases Proco and Eddy we analyzed role switching and found that it did not happen very much, if at all. However, on the contrary to the reports from literature, this did not seem to passivize the navigator. Of course in case Eddy, the navigator sometimes left the PP session if things got very simple, and in case Proco PP was used only for tasks that were difficult or when the purpose was explicitly to teach a less experienced partner. Proposals from literature are often from contexts where PP is intended to be used for everything. I may certainly be true that sitting as a navigator for a long time when the driver writes trivial code makes the navigator passive.

In all our cases the attitudes towards PP were good. There were practically no developers who were against using it, and some developers even preferred it over solo programming. In case Proco we analyzed also developers preconceptions to PP and found that the attitudes become much better after developers have really used PP.

Our cases support the proposed benefits of PP. In case Casino we measured the effects of PP objectively, and in the other two cases analyzed them based on the perceptions of the developers. In all cases there seemed to be positive effects for knowledge transfer/learning, quality, enjoyment of work and team spirit. The negative effect of clearly increased effort for implementing tasks seemed to realize only during the learning period of PP (case Casino), or when doing trivial tasks (case Eddy). From the viewpoint of overall productivity of an organization, some amount of effort increase related to individual tasks can be accepted, because the realized benefits are likely to compensate this in the long term.

Case Proco brought up certain challenges for the adoption of PP. It differed from the other cases, firstly because they were large organization and secondly because they aimed only for a limited amount of PP use. Noise and work space issues in an open office had to be solved to make PP a proper practice in the organization. Organization of PP required also more attention than in the two other cases, where PP was used simply for practically everything by everyone. Especially assigning tasks for individuals hindered allocating resources for PP, because task owners naturally prioritize finishing their own over helping other developers with their tasks when in a hurry.

References

Beck, K., Extreme Programming Explained, Addison-Wesley, 2000.

Begel, A., and Nagappan, N., "Pair programming: What's in it for me?" Proc. 2nd International Symposium on Empirical Software Eng. and Measurement (ESEM), 2008, pp. 120–128.

Gallis, H., Arisholm, E., and Dybå, T.,"An Initial Framework for Research on Pair Programming", In Proceedings of International Symposium of Empirical Software Engineering (ISESE), 2003.

Hannay, J.E., Dybå, T., Arisholm, E., and Sjøberg, D.I.K., "The effectiveness of pair programming: A meta-analysis," Information and Software Technology, vol. 51, no. 7, 2009, pp. 1110–1122.

Nosek, J., "The Case for Collaborative Programming", Communications of the ACM, 41(3), 1998, pp. 105–108.

Salo, O., and Abrahamsson, P., "Agile methods in European embedded software development organisations: A survey on the actual use and usefulness of Extreme Programming and Scrum," IET Software, vol. 2, no. 1, 2008, pp. 58–64.

Schindler, C., "Agile software development methods and practices in Austrian IT-industry: results of an empirical study," Proc. Int'l Conf. Computational Intelligence for Modelling, Control and Automation (CIMCA '08), 2008, pp. 321–326.

Williams, L., and Kessler, R., *Pair Programming Illuminated*, Addison-Wesley, Boston, 2002.

Wilson, J., Hoskin, N., and Nosek, J., "The Benefits of Collaboration for Student Programmers", In Proceedings of the 24th SIGCSE Technical Symposium on Computer Science Education, 1993, pp. 160–164.

7 Challenges and Guidelines for Defect Reporting

7.1 Introduction

When a defect is found, the findings are reported to the person responsible for fixing defects. This reporting activity is called defect reporting.Based on the report the reporter has constructed, the software developer tries to understand the technical causes of the defect and ultimately fix them. Finding the technical causes of the defects and fixing them is called debugging.

First, a defect reporter creates a defect report. In it, she describes briefly what she tried to do with the software and how the software failed in doing that it was supposed to do. The software developer issued to the defect report reads the report and verifies that she understands the defect by reproducing it in her development environment. After reproducing the defect, the developer can use a variety of debugging techniques to find the root cause of the defect and fix it.

Sometimes the developer can reproduce the defect right away. However, this is not always the case. Some defects are simply difficult to reproduce and sometimes the developer just does not have acquired enough information about the defect to reproduce it. The problem is that the defect reporter does not know what the developer would need for being able to fix the defect.

There are two strategies which can be used to report defects:

1. The reporter tries to figure out what information is needed to fix the defect.
2. The reporter tries to report all the possible information related to the defect.

The strategy 1 is the most common in practice, but results to insufficient defect reports and lot's of time is spent during the defect fixing to collect the missing defect information.

The strategy 2 is not widely used as requiring a large workload from the reporter, or a sophisticated defect reporting infrastructure. It is also difficult to define the “all the possible information related to the defect”. On the other hand, the advantages include less effort required to solve the defects and the defect repositories contain more accurate and complete defect reports. In the strategy 2, the issue of large workload of the reporter can be overcome by collecting the needed information automatically. This is what we studied by a survey during the ESPA research project (Laukkanen and Mäntylä 2011). We asked the developers of five software companies to evaluate the defect reporting in their companies through several questions. Next, we will introduce the main results of our survey and the practical guidelines we would like to pinpoint from the results.

7.2 Results of the Survey of Automatic Defect Reporting

7.2.1 What information do developers consider useful for fixing defects?

We asked which pieces of information the developers considered useful for fixing defects. The developers rated the following items the most useful:

Steps to reproduce are the steps that the developer has to go through in the software to reproduce the defect. It helps the developer to understand what the defect is and how to start debugging it.

Part of the application is the part where the defect occurs, for example, the print dialog. With part of the application, the developer can get a quick understanding where the defect is from the viewpoint of the user and can understand the context of the remaining defect report.

Observed behavior is the behavior that the reporter observes, when the steps to reproduce are executed. Tightly connected to this piece is the **expected behavior**, which tells us what the reporter expected the software should do. This is important because it is not always obvious what should be the result.

Screenshots are images about what the user sees from the software when the defect happens. This is useful for the developer, because it instantaneously links the defect report to the software in question. Also recording the usage of an application was reported to be useful for fixing defects.

Operating data is the data that the user is handling when she is using the software. For example if a user is doing photo manipulation, the image she is manipulating when the defect occurs is the operating data. This can be useful for fixing the defect, because the developer can directly try to

reproduce the defect with the same data. The defect might be happening because there is something exceptional about the data that is being operated.

Configuration of the application covers all the settings of the software. The configuration affects the defects in the same way as the operating data. The defect can be tightly coupled with the configuration; the developer fixing the defect might not be able to reproduce the defect if the configuration is not exactly the same as when the defect occurred for the user.

All the items listed are useful almost in any software context. However, there are also items that depend on the context. Such items are, for example, database logs and website address. Collecting context specific information to the defect reports should be also taken into consideration, because they might be essential information for fixing the defects.

7.2.2 How easy would it be to collect the useful information automatically?

The developers who participated in the survey also rated the items based on how easily they could be collected to the defect reports automatically. Next we will discuss how easily the most useful items could be collected.

Steps to reproduce could be recorded, for example, with a video capture tool or recording the user interaction with the software. Sometimes it can cause quite a large overhead for using the application. However, for testing purposes recording the application usage should be feasible. It is also important to define steps to reproduce in textual form, because defects are often searched by words from the defect database.

Part of the application could be collected automatically with a screenshot or by collecting the webpage address. For example, in Google Chrome, defects can be reported with a GUI tool that takes a screenshot automatically and also collects the webpage address to the defect report. However, part of the application can be difficult to even determine if the software is server-side software. Thus, it is rather software-specific if the part of the application can really be collected or not.

Observed behavior is, like the steps to reproduce, possible to collect automatically by taking a screenshot, if the behavior is static. We still recommend that it should be also described by words for text searching. The **expected behavior** cannot be collected automatically, because it depends on the user of the application.

Screenshots could be and are collected easily to defect reports automatically.

Operating data could be collected to defect reports automatically, but the data can contain confidential elements. The reporter could try to obfuscate the data, but that requires additional workload and therefore is not often a realistic option.

Configuration of the application could be, like the operating data, collected to defect reports automatically. However, it can also contain confidential information in some cases.

Context specific information need to be defined in the defect reporting tool for automatic collection. If context specific information can be acquired by shell commands or by reading log files, collecting them can be straight-forward.

In conclusion, there are just screenshots out of the useful items that can be collected automatically rather straight-forwardly. In the next section, we form guidelines based on these and other results in the survey.

7.3 Guidelines Based on the Results

In this section, we discuss about a crash reporting which is a special case of defect reporting, review data collection tools and methods for defect reports and lastly highlight the importance of user input for defect reports.

7.3.1 Crash reporting

Crash reporting is an exceptional case of defect reporting, because it can be completely automated. For example, Windows Error Reporting (Glerum et al. 2009) collects crash reports for any Windows application. Crash data contains debugging information, usually a core dump and additional information about, for example, loaded libraries. A crash also happens in a very specific part of software, which can be identified with a **stack trace**. Usually by acquiring crash data the developer can reproduce the defect.

There are multiple frameworks for automatic crash reporting, such as Windows Error Reporting, Breakpad, CrashRpt and EurekaLog. These tools can be very useful when the software is deployed to the user environment and there are multiple variables and library versions that could not have been tested during the development. Still, a manually started defect reporting procedure should be implemented also, to acquire defect reports also from defects that do not cause the software to crash. This is especially important during the development of the software, when the defect reports can be feature requests, but have same elements as defect reports.

7.3.2 Data collection tools

Data collection methods, such as screen and video capturing tools can be used to quickly demonstrate where the defect is and how it is reproduced, even if the steps to reproduce are particularly complicated. Data collection should be seamless to the defect reporting system. Other data collection methods would be capturing key strokes and mouse movements, to practically see everything that the user is doing with the software. Log files and other files and application information should be collected also, but they need to be defined application specifically.

User input is important. Despite all the data collection methods, clear textual user input should be collected, whether or not it is a crash or a feature request. User input drives two basic functions:

1. It is natural for the reporter to report what the software did and did not by using natural language.
2. Defect reports are searched frequently for duplicates, so text-based information should be saved with the defect report.

7.4 Summary

To summarize previous sections, there are three main concepts about automatic defect reporting:

1. Integrating automatic defect reporting to the software in question to gain detailed information about the software.
2. Recording or capturing the usage of the software by video recordings or screenshots. This will allow the developer to quickly focus on the situation in hand that the user is facing.
3. Reporting both collected information and user given information to the defect report and sending that information to the developer.

To have effective and comprehensive defect reporting practices, one has to assess all these points. It depends highly on the software in question how largely defect reporting practices should be taken into account. In general purpose software, the reporting procedures should be mature, but for specific software that is seldom updated, it might be too much work to implement a fully operational automatic reporting infrastructure. In all cases the software as a whole should be taken into account and it should be carefully assessed that why the defect reporting is done in the first place and can there be any communicational problems with it.

References

Glerum, K. et al. 2009. Debugging in the (Very) Large: Ten Years of Implementation and Experience. In Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP).

Laukkanen, E. I. and M. V. Mäntylä. 2011. Survey Reproduction of Defect Reporting in Industrial Software Development. In Proceedings of the 5th International Symposium on Empirical Software Engineering and Measurement (ESEM).

8 Appendixes

APPENDIX A: Analyzing development process using TQP Framework

In this section we present an example of how to use the TQP Framework for analyzing and understanding quality practices as part of incremental development process. As an example development processes we took four agile software development methodologies and use the framework to understand how the quality practices work in those methodologies. These methodologies are eXtreme programming (XP), Feature Driven Development (FDD), Dynamic Systems Development Method (DSDM), and Microsoft's Synch-and-Stabilize. We analyze two of the methodologies, XP and Synch-and-Stabilize, in more detail and identify possible weak spots in those processes from the quality viewpoint.

In-sync quality practices are not delayed to any later testing phase. Instead, these activities are performed during implementation, as part of the design and coding tasks, regardless of whom, e.g., a developer or tester performs these tasks. These activities give instant feedback to developers and therefore help drive development in the right direction. In agile methods, the in-sync quality practices, operating in heartbeat time horizon rhythm[3], are very strong as can be seen in Table 20. Developers' practices are particularly comprehensive and rigorously defined in XP, but the other methods also put a strong emphasis on unit testing, code and design inspections, regular builds and short integration cycles, which are all practices in the developers' everyday design and implementation work. These practices work as a strong basis for the agile development process and strive for ensuring that good enough quality is produced out of every development task.

[3] The quality practices can be connected also to the different time-horizons. Typically the in-sync practices operate in the rhythm of heartbeat time horizon and the off-sync practices are performed either for each iteration or only generally for the entire release project. See the part I of this book for more detailed descriptions of the time horizons and the CoC framework.

In agile methods, the off-sync quality practices are much fewer than the in-sync practices. In Table 20 we can notice that ins some methodologies only one or two off-sync practices can be found for ensuring and evaluating the quality of the produced software increment. In addition, these practices are rather superficially defined compared to the in-sync time horizon practices. Some methods, e.g. XP, trust almost purely on the strong in-sync practices and only leave progress tracking by evaluating the acceptance test results as an off-sync practice on the iteration time horizon. The other methods that have less rigorous heartbeat practices recognize the need for evaluating the achieved quality on the iteration time horizon by system testing, but do not give concrete guidance on how to perform it as a part of the process. For example, in FDD the only advice given for accomplishing this is to decide which builds, and how often, are handed over to separate system testing. The DSDM method has a stronger approach to off-sync quality practices and not so detailed guidelines for the in-sync practices on the heartbeat time horizon. The approach is like a small waterfall process inside each iteration, or time-box, as it is called in DSDM.

In the next two subchapters we describe and analyze the quality practices of two of these approaches, extreme programming and MS Synch-and-Stabilize, in more detail.

Table 20 Agile quality practices classified by rhythm

	XP	FDD	DSDM	Synch and Stabilize
Off-sync practices	Evaluating the acceptance test results Architectural spikes	Separate system testing	Contractual acceptance testing (if required) Integration, system and acceptance testing inside each time-box User testing Evolutionary prototyping Document reviews	Release stabilizing phase 20% refactoring tax Beta testing Increment stabilizing phase Test releases and test release document Using the internal releases in development Battery of performance and other types of test suites Design reviews and checklists Bug Bashes
In-sync practices	Test-driven development Continuous integration (daily-builds) Pair programming Acceptance tests Collective code ownership Coding standards Simple design & continuous refactoring On-site customer	Unit testing Regular builds Design inspection Code inspection Individual code ownership	Unit testing Reversible changes Active user involvement	Daily builds Buddy testers Testing the private build Quick smoke tests for private build Continuous refactoring Debug code Code reviews Usability testing

XP

Extreme Programming (XP) (Beck 1999, Beck 2000, Jeffries et al. 2001) emphasizes short iterations, team-work, continuous customer involvement and producing only artifacts with immediate value. The quality practices of XP in Table 20 consist mainly of developers' in-sync practices. Thus, the QA

approach of XP is mainly based on relying on low-level practices that are rigorously followed in all development work.

In XP unit tests must be created before the actual implementation code. This is called test-driven development. XP also requires that all code, including the unit tests, must be written in pairs. Every time pairs are about to implement a task, they must make sure that the end result represents a design that is as simple as possible. As XP does not emphasize big up-front-design, the developers must also put effort on continuous refactoring to keep the system simple. After a task is finished the pair that implemented the task runs all the unit tests and if there are no problems, the code is immediately integrated into the repository. Thus, XP focuses on the following implementation cycle: pair up with other developer, implement unit tests and figure out a simple design, implement the feature, refactor if needed, make sure all unit tests pass, and integrate the produced code back into the repository. Other in-sync activity performed at heartbeat rhythm is the creation of acceptance tests. The customer should take the responsibility of creating the acceptance tests for each feature as features are completed. Finally, the on-site customer is needed to advice the developers in interpreting the user stories that are used to capture the customer requirements.

There are also in-sync quality principles that support the heartbeat level implementation cycle presented above. Collective code ownership that allows anyone to change any code at any time makes sure that the progress is not dependent on individuals. Coding standards keep the code readable and make sure all developers can read each other's code.

There seems to be only two off-sync quality practices that aim at controlling development at the iteration time horizon. Acceptance test results are used to track iteration progress, i.e. to demonstrate the achieved benefits. Customer is responsible, in collaboration with the tester(s) and developers, for providing acceptance tests for each implemented feature. These tests are used to track how the iteration is proceeding as well as to show that the implemented features exist and work as the customer had specified. The second off-sync practice is architectural spikes that are used for prototyping risky product areas in order to mitigate risks before the actual implementation and make better estimates on how long it will take to implement a feature. Although spikes concern a single feature, they are rather an off-sync planning tool than an in-sync implementation task.

Synch-and-stabilize

As a second detailed example of analyzing an IID methodology with the TQP framework we present Microsoft's Synch-and-Stabilize process (Cusumano and Selby 1998). In the Synch-and-Stabilize process, a release project is divided into several phases: a 3-12 month planning phase, a 6-12 month development phase that is executed in three iterations, and a 3-8 month stabilization phase. The quality practices of the Synch-and-Stabilize process are summarized in the Table 20.

In the Synch-and-stabilize process, there are a lot of in-sync quality practices both for developers and specialized testers. In the Microsoft's development organization in question, there were roughly as many testers as developers. The testers are paired with individual developers, meaning that each developer has an own buddy tester. A buddy tester reviews specifications, writes test cases, and tests the developer's private builds before the code is checked into the common repository. The developer also tests private builds manually and with automated smoke tests to detect side effects that the change might cause to other functionality. After that, the buddy tester tests the new features with the developer's private build before integrating the code into the common code base. This integration happens after each feature is implemented, which could range from one day to a few weeks. The daily build practice is used to build the whole product every night and execute an automated test suite.

Continuous refactoring is also an essential in-synch practice of the Synch-and-Stabilize process. This means that developers continuously rewrite portions of existing code, without changing the behavior, to make the code better. Microsoft estimated that 50% of all source code becomes rewritten in every 18 months.

The features in Microsoft's desktop application products are very user interface intensive. This has led Microsoft to include an in-sync usability testing practice tightly on the heartbeat rhythm. This means that every feature that affects the end users is tested in a usability laboratory already when the developer is finishing the implementation of the feature. This makes usability testing an integral part of development and enables almost instant usability feedback to developers.

Other in-sync quality practices that are performed in the heartbeat rhythm are code reviews and writing of debug code. Code reviews are used informally and mostly for knowledge sharing and teaching purposes as well as for checking the work of new developers. Developers write debug code with several runtime checks to facilitate testing and to boost the debugging process.

The Synch-and-Stabilize process also contains several off-sync quality practices. First, a 2-5 week iteration stabilizing phase is utilized to ensure that the required quality level can be achieved and a completed set of stable enough functionality will be ready at the end of an iteration. After that period, yet another 2-5 weeks of buffer time is allocated for unexpected issues that always arise during a development iteration.

Handing over regular test releases for system testing is clearly an off-sync practice. Selected daily builds are handed over for testing once a week. Each such test release is accompanied with a test release document that describes the new and changed features in the release, how the new features should work, and ideas for how the features could be tested. Testers test the new features in the test releases continuously throughout the project, not just during the stabilizing phase. In addition to releases for testing they also use the intermediate versions of the product in the development organization. This practice of “eating your own dog food” is used frequently, but is not included as an in-sync practice, because the versions are not taken into internal use and feedback not collected in the heartbeat rhythm.

The specific off-sync testing practices is using a battery of several different test suites and methods that testers continuously use to test builds during each iteration to find critical problems. The test types include e.g. performance testing, scripted testing, and gorilla testing.

Off-sync practices also include design reviews and checklists that help to assess if a certain milestone or iteration goal has been reached. The design reviews are formal, intensive and used at the end of the planning phase before the actual development iterations start.

An interesting off-sync practices is so called “bug bashes”. This means a session or party where the whole development organization comes together and does ad-hoc, or exploratory, manual testing in a bug finding competition.

Some of the off-sync quality practices of the Synch-and-Stabilize process operate on release time horizon. These are release stabilizing phase, beta testing, and a 20% refactoring tax. The release stabilizing phase is a similar practice as the iteration stabilizing phase. It means that time for a 3-8 month stabilizing phase is allocated after the last development iteration to stabilize the product: test, debug, and fix defects. The release stabilizing phase also includes buffer time for unexpected issues and changes. Beta testing is performed during the release stabilization phase by delivering beta versions to thousands of beta customers who test the product and report defects and problems back. The last off-sync practice is the 20% tax for refactoring the code. This practice refers to Microsoft’s explicit decision to use 20% of their development effort in each release for rewriting the

existing modules or features in order to maintain the internal consistency and quality of a product. This time is often spent during the initial planning phase of a release project when the development iterations of the new release have not yet started.

Analysis of the Three Viewpoints

With the help of the TQP Framework, we can see that the developer oriented agile methodology XP relies mostly on the in-sync quality practices. Microsoft's Synch-and-Stabilize development process is more comprehensive, and includes a wider range of quality practices, as can be seen in Table 20.

From the purpose of the practice viewpoint the practices of XP mainly seem to fall into the *building in quality* category. Only the acceptance testing and on-site customer are practices for *evaluating the achieved quality*. The Synch-and-Stabilize process, instead, includes numerous practices for evaluating the quality. Such practices are beta testing, test releases, using internal releases in development, performance and other type of test suites, design reviews, bug bashes, buddy testers, testing the private build, code reviews and usability tests. This analysis shows that the variety of quality practices in Synch-and-Stabilize is wider than in XP, in terms of the core purpose of the used quality practices.

The second viewpoint in our framework is the attitude. The attitude of the XP's quality practices is mostly constructive, *demonstrating the achieved benefits*. The core XP practices, test-driven development, pair programming, continuous integration, and even the acceptance testing, are constructive by nature. The aim of the practices is not to focus on pointing out problems and digging out all possible problems, instead, the practices aim to building confidence and demonstrating that the implemented features are done and work as desired.

From the third, roles, viewpoint it is clear that Synch-and-Stabilize includes a very significant testing effort that is recognized as a separate function performed by dedicated testers in collaboration with developers. This is clearly visible as off-sync quality practices, which include many practices of dedicated testers. The stabilizing phases are visible both on iteration and release time horizons. These phases give testers the required time interval to exercise the product extensively without developers concurrently producing new, fragile features. The dedicated buddy testers in Synch-and-Stabilize process are a good example of how the independent role can work also in-sync with the development. In XP, on the other hand, the developers have a big responsibility of the quality practices, the tester

role is part of the XP, but it is not very clearly described. In XP only the acceptance testing practice that mainly is the responsibility of customer and tester(s), is at least to some extent independent from the developers.

After further analysis of the quality practices, it seems that the purpose of the in-sync practices is mostly to build in quality and the attitude is usually constructive. The practices concentrate on demonstrating the benefits, and focus on the level of individual features. Furthermore, the in-sync practices are typically targeted for developers. Evaluating the achieved quality and applying destructive attitude are traditionally tasks of specialized testers, and from the methodologies analyzed here are only visible in Synch-and-Stabilize. It seems to be hard to introduce in-sync practices that evaluate the product from the testers' destructive perspective down to the heartbeat time horizon to work synchronized with the development tasks. This shows clearly in XP that lacks practices that would reveal product risks by exercising the whole system from different perspectives. Without these practices, it can be hard to control the development project, even if the project is carried out in small iterations.

An interesting detail that the TQP Framework reveals in this analysis is how the same practice can be applied with different rhythm in the same process: the refactoring practice is used as in-sync practice in both XP and Synch-and-Stabilize. As an in-synch practice refactoring is applied continuously by rewriting code immediately when poor structures are encountered or a new design is needed. However, Synch-and-Stabilize includes the refactoring practice as off-synch practice operation on the release time horizon as well. Here, the practice (20% refactoring tax) is applied for larger refactorings during the planning phase of a release project. Anther example of a practice that can be applied both in-sync and off-sync is reviews. Identifying different instances of the same practice helps understanding how and when to apply the practice and what is the purpose of applying the practice in each context.

Analyze the quality practices in your own project using TQP Framework

- Divide the practices to in-sync and off-sync practices. Can you associate the practices into certain time horizon (heartbeat, iteration, release)?
    - How could you improve the feedback and quality tracking?
    - What new practices could be utilized in different time horizons?

- What current off-sync practices could be made in-sync practices? Why?

- Analyse the practices from the three viewpoints of the framework
  - Building in quality vs. Evaluating achieved quality
  - Demonstrate achieved benefits vs. Point out problems
  - Developers vs. Independent testers
- What other improvements should be made?

Appendix B: Good practices of an Intelligent Tester

In this appendix we present descriptions of the 22 practices of an intelligent tester that were identified in our research. These practices are based on the findings of an observation study where testers' actual work was observed in companies. The practices are presented according to the classification presented in section x.y.

In addition to classifying the practices into these categories, one or two main purposes of each practice are marked using the following notation:

 A technique to guide exploring

 A technique for achieving or tracking coverage

 A technique for evaluating the results of a test

 A defect detection strategy

Exploratory Session Strategies

User interface exploring	
Description	Structure exploratory testing based on the user interface components and visible features. Test each component or feature by free exploratory testing or by using some other IMT Practice. Use the user interface for giving the structure to follow when you proceed in the testing, but do not restrict the test ideas and your thinking on the GUI layer only.
Goals	Covering all user interface features.
	Identifying missing, extra, and wrong GUI features.
	Regression testing old functionality.

Exploring weak areas

Description	Explore areas of the software that are weak or risky, based on some experience or knowledge. Apply free exploratory testing or some other IMT practice by focusing to estimated weak spots of the software. Such weak spots are all areas that are somehow known to be risky: - have been weak in previous testing - have been challenging to design or implement - have a lot of customer complaints - are complicated - were coded in a hurry - have lots of changes - based on coders' opinion - based on testers' opinion - pure hunch Focus testing to most risky areas of the software. Reveal important problems efficiently.

Aspect oriented testing

Description	Test for a certain aspect through the whole software or some part of the software. Examples of such aspects could be checking for correct texts, consistency of the GUI, or verifying a certain feature, e.g., undo feature, throughout the application. Guiding exploratory testing by focusing to a certain aspect with the goal of testing the aspect throughout the software. Covering a certain aspect throughout the software.

Top-down functional exploring

Description	Test first on higher level with typical cases and simple checks. Proceed gradually deeper in the details of the tested functionality and use experience to guide your testing. Select things to focus and issues to explore in detail based on your experience and understanding of the features and implementation as you gain it during the testing. Proceed into deeper details step-by-step. The following questions give and example of the information that each step can provide: - Is this function implemented? - Does the function do the right thing? - Is there missing functionality? - Does the function handle the special cases? - Does the function work in combination with other functions and together with the rest of the system? To get first high level understanding of the function and then deeper

	confidence on its quality set-by-step. Utilizing tester's experience in guiding where to focus and what detail to test most thoroughly.

Simulating a real usage scenario

Description	Simulate the real use of the system. Explore through the functions by following a realistic scenario and performing tasks that the real user would do in such situation. Cover the real scenario entirely, not just some functions or part of the normal flow of users work. You can start with a simple basic case and then take the scenario further by designing a complicated scenario and even further by so called "soap opera" scenarios. Soap opera scenarios are complicated scenarios where all aspects of the scenario are exaggerated. Guiding exploratory testing to follow a realistic usage scenario. Revealing defects that affect the user in realistic usage scenarios. Understanding how the system supports users real working tasks, not just individual features.

Smoke testing by intuition and experience

Description	Select small amount of features for smoke testing based on experience and intuitive hunches. Can be applied as a separate task or as a continuous practice of performing smoke tests for varying set of features along other testing activities. Guiding exploratory testing to evaluate the test readiness of a new build. Revealing the most obvious defects, changes to existing features and assess overall robustness.

Documentation-Based Session Strategies

Using data as test cases

Description	Use a pre-defined test-data-set as test cases. The data includes all relevant cases and combinations of different data and situations. In this practice, a lot of domain knowledge is captured in the test data in order to ensure it is covered and available in testing. Using this data the tester can manage the coverage of testing by covering all relevant data sets when testing each function or feature. The functions and features are tested exploratory, but the pre-defined data set is used to achieve systematic coverage of the relevant contexts and combinations in which the functions must operate. This practice is especially suitable for situations where data is complex, but functions are simple to perform. Another applicable context is when creating the test data requires much effort or specific skills. An example of this practice is a CRM system where the test data represents the different kinds of customers with different properties and situations. This set of customers is then used for testing each function or feature of the system. Managing coverage of testing by pre-designed test data that captures relevant details of the applications domain as well as complicated variations and combinations of the data. Documenting the relevant details of the application domain in the test data.

Exploring high-level test cases

Description	Use pre-designed high level test cases (or "test ideas") to guide and structure exploratory testing. The detailed test steps and inputs are designed exploratively during the testing. The practice can also be used to apply the same high level test idea efficiently to exploring multiple features or places in the software. Plan and track exploratory testing by high-level test cases. Revealing defects by applying the pre-designed test ideas. Covering the designed high-level test cases.

Checking new and changed features

Description	Check new and changed features against release notes or other specifications by free exploration or by using some other IMT practice. Verify defect fixes based on the defect reports. Plan, track and give structure for exploratory re-testing. Cover all changes or fixes. Check that features work and changes are implemented or defects fixed as described.

Simulating abnormal and extreme situations

Description	Test how the system behaves in a certain abnormal or difficult situation. Some problematic situation or risk that is identified earlier is comprehensively explored and it's consequences or impact analyzed. Another way of using this practice is trying to cause a potential failure by simulating a certain exceptional case. Providing understanding and information concerning some specific problem or risk. Show that a certain exceptional situation can cause a failure.

Testing alternative ways

Description	Identify and test all different ways of performing the same function. Identify defects and inconsistencies between the alternative ways. Find out missing or incomplete alternatives and excess functionality. Typical examples of such alternative ways are menu, context (pop-up) menu, shortcut, keyboard, options, command line arguments, etc. Apply the same function to objects of different types or sizes. Identify differences and defects in functions when applied to different objects. Typical examples of such objects: - Selections, e.g., nothing selected, part of an item, one item, many items, all items. - Default target, selection, whole document - Applying function locally or globally Hierarchy: levels, types of objects in the hierarchy Revealing defects and differences that are related to the different kinds of targets of a function. Guide exploring a function with different objects to cover all types of objects the function can be applied to.

Exploring against old functionality

Description	Explore the functionality of a changed or new implementation in relation to the older version of the software. - Is all old functionality available? - What has been changed? - What new opportunities there are? - Can new things cause problems? Identifying and analyzing differences compared to earlier functionality. Understanding how a changed or extended feature should actually work.

	Revealing defects by identifying unintentional or badly designed changes in the functionality or results.

Persistence testing

Description	Test that data is correctly stored in different scenarios of normal operation and that data is not lost or corrupted in exceptional situations. These situations are usually related to saving data, canceling operations and exiting the system in combination with different types of changes, and modifications to the data. Exceptional aborts, resource failures, or partially completed transactions are of a special concern. Reveal defects that are related to storing data and maintaining the persistence and integrity of the stored data. Guiding exploratory testing to focus revealing possible problems with the persistence or data integrity.

Feature interaction testing

Description	Test unintentional interactions as well as known dependencies between the functions. These interactions and dependencies can occur inside a single feature or as combined effect of two or more features. This practice is based on hypothesis that the features could affect each other in harmful ways. The tester should, based on experience and understanding of the system, reveal problems caused by these interactions. This is a practice that is applicable, e.g., exploratory regression testing where the tester's experience and understanding is utilized for revealing interaction defects, instead of a large (automated) set of regression tests. Reveal defects and problems that are caused by interacting features in the software. Guide tester to focus on these interactions and utilize one's experience to revealing the problems.

Defect based exploring

Description	Explore variations and dependencies of a certain defect or defect group. This exploring can be based on a found defect or an assumption of typical defects. Based on the base defect explore other places where the same or similar defects might occur, or if there are variations of the same defect. Another viewpoint is to analyse if a single defect is in fact only a symptom of a larger or more general problem in the software. For example, if removing an attribute in some data structure is possible when it should not be, you could explore if it is actually possible also to modify, move, or add an attribute when it shouldn't be. Reveal full extent of a defect or apply one defect as a failure model to reveal other similar defects or problems in other areas of the software.

Testing to-and-from the feature	
Description	Test all things that affect to the feature and all things that get effects from the feature. Systematically cover the feature's interactions. Reveal defects that are caused by a not-the-most-obvious relationship between the tested feature and other features or environment. Continue exploring a certain scenario or feature by analyzing what is the next thing user would do. Proceed to test the next thing and ensure that the system works correctly in scenarios where users related tasks follow each other. Identifying problems that the user encounters when the features of the system are used together and functions should work in situations that the other functions leave the user. Revealing problems that incomplete or incorrect features cause, but are revealed only later in the following steps of the users work task, even completely outside of the tested system.

Comparison Techniques

Comparing within the software	
Description	Compare similar features in different places of the system and test their consistency. If the same function can be performed with different ways or mechanisms in the same system, compare the results of the function. Revealing problems in consistency of functionality inside the software. Helping decide if a feature works or is implemented correctly or not.

Comparing with another application or version	
Description	Compare the features and implementation between the applications of the same product family or with competing products. Use the comparison to analyze how things should be done in this product and understanding if a certain issue actually should be considered a defect or not. Identifying consistency problems within a product family. Understanding the correct way of working by comparing to how similar features work in other products. Identifying shortcomings in comparison to competing products.

Checking all the effects

Description	Ensure that the function or feature has correct effect in all places of the system it should. Check results and effects of one feature or setting to other features, and working of the system in different situations. Think hat are all the places and situations where the information that the feature affects is used or displayed. Think also side effects of the function, what other things should happen when this action is performed? For example, setting certain attributes should cause other attributes to be set automatically. Guide understanding the full effect and consequences of the tested feature and determine correctly if the feature works or fails.

End-to-end data check

Description	Check that the data is correctly used and viewed on all layers (e.g., GUI, business logic, database). Check that the data propagates correctly in both directions. Test for correct handling of possible modifications to the data that are not caused by the tested feature or system. This practice can involve, for example, checking the results of a test straight from the database using sql editor as a tool.
Goals 	Making sure that all details of the features work completely. Focus is especially on effects that are not directly visible through GUI. Checking that the graphical or other interfaces shows correct data. Revealing defects in propagation of data through the system.

Input Techniques

Testing boundaries and restrictions

Description	Test boundary values of input data. Cover all explicit and implicit restrictions of the data and functions. Find out if there are ways exceed the stated or implicit limits and restrictions of the system and test beyond those limits. Revealing defects that are associated with handling boundaries and restrictions. Focusing exploratory testing to boundaries and restrictions.

Testing input alternatives	
Description	Test each alternative for each input. Identify relevant alternatives that affect the behavior of the function or are likely to fail and reveal a defect. Cover these alternatives systematically. Focus on the differences in functions and other effects that the different alternatives should have. Managing the coverage of testing single function or feature. Covering all equivalence classes of individual inputs.

Covering input combinations	
Description	Identify relevant combinations of the inputs, outputs, data, etc. that affect the functions and behavior of the system. Cover systematically the identified combinations by testing and focusing to the specific effects of the combined variables and any unexpected side effects. Covering systematically the combined effects of two or more variables. Checking that combinations result to correct behavior. Finding defects related to combined effect of several inputs.

Appendix C: Classification instrument of problem causes

The process areas of causes

Process area	**General characterization of the detected causes**	**Concrete examples of detected causes**
Management (MA)	Company support and the way the project stakeholders are managed and allocated to tasks.	The quality of the product is low prioritized in the company, really. Lack of steering the projects and their related interactions.
Sales & Requirements (SR)	Requirements and input from customers.	Too high number of customer requests. It is presupposed that the developer understands an ambiguous specification.
Implementation Work (IM)	Coding of new features and defect fixing.	The implemented interfaces are non-functional. The features are implemented without caring their quality. Too much unreported error handling.
Software Testing (ST)	Causes are focused on software testing.	No schedule for software testing. Tests are not conducted against proper requirements.
Product Release and Deployment (PD)	Releasing and deploying the product.	Product installation is experienced difficult. Developers do not configure the things they have implemented.
Unknown (UN)	Causes that cannot be focused on any specific process area.	Laziness. Gratuitous work is done.

The cause classes and related types of causes

Class / Type	**General characterization of the detected causes**	**Concrete examples of detected causes**
People	This class includes the people related causes	
Instructions & Experiences	This type includes causes of missing documentation and lack of experience. The needed documentation is missing or inaccurate, and the lack of experience complicates the work.	Lack of instructions when and how to verify. No knowledge on how many files have to be modified to configure.
Values & responsibilities	This type includes causes of bad attitude and lack of taking individual responsibility. The people do not care about important things and they look out for number one.	People do not care if the number of bugs increases. We have a problem in our organization culture.
Co-operation	This type includes causes of inactive, inaccurate, and missing communication between the stakeholders. The people do not communicate actively or share knowledge on their own will.	The requirements were not inspected with large enough group. Miscommunication between the developers and testers / project engineers.
Policies	This type includes causes of not following the company policies.	The feature is marked as "finalized" without testing it. New issues are not recorded to bug repository even when they should be.

Tasks	This class includes the task related causes	
Task Priority	This type includes causes of task priority. The priority is missing, wrong, or too low.	New functionality is more important than the product quality. The priority of issues is too low.
Task Output	This type includes causes of low quality task output. In our terminology the task is a general term which corresponds the tasks of all stakeholders, e.g. the managers may do inadequate resource allocation whereas the developers may do bad code, etc.	Requirements are insufficient. Management work is insufficient.
Task Difficulty	This type includes causes of challenging tasks. The task requires too much effort, time, or it is too difficult.	Creation and implementation of standard tests is too difficult. It is difficult to create a comprehensive specification.
Methods	This class includes the methodological causes	
Work Practices	This type includes causes of lack of current working methods. The method is missing or inadequate.	Regular unit testing was insufficient throughout the project. Development work is done directly to the testing environment.
Process	This type includes causes that are focused on the current operations model. The model is unclear, vague, too heavy, or inadequate.	Systematic testing is not done as a regular process. Product versions are developed too long period in parallel.
Monitoring	This type includes causes of lack of monitoring. The management does not know the project status caused by the lack of monitoring the progress.	Unrealistic understanding of the quality during the development work. The installations are usually scattered and nobody knows which one is in use.
Environment	This class includes the environment related causes	
Existing Product	This type includes causes of the existing product, which is too complex and the old low-quality code creates challenges.	The structure of the product is decayed during the past. Nobody knows the whole product as it is an extremely large system.
Resources and Schedules	This type includes causes of wrong resources and schedules.	Lack of time in testing. Lack of time to define the issues specific enough.
Tools	This type includes causes of missing or insufficient tools.	The requirements template does not require speaking out for constraints. The version control system does not support developing customer specific functionality.
Customers & Users	This type includes causes of customer requests and users expectations and needs.	The customers' desires are not analyzed and prioritized quickly. What is important for the customers is not mapped well.