Ensuring the pigs don’t chicken out

Improving the use of good practices

in agile software development

Roel van der Hoorn

Ensuring the pigs don’t chicken out

Improving the use of good practices

in agile software development

Author: Roel van der Hoorn

E-mail: vanderhoorn@gmail.com

University: University of Twente

Master: Business Information Technology

Date: July 1, 2011

Summary

This research takes place in a small company in Enschede, The Netherlands. The company delivers web-related IT services and software. Its main activities are web development and web hosting. The reason for this research is that managers and developers are not always completely satisfied with the software development process. The following research goal is applicable:

> The purpose of this study is to *improve* the *agile software development process* from the point of view of *developers and managers* in the context of *a small web development company with off-site customers* by analyzing *available documents*, doing a *literature review* and conducting *interviews*.

This lead to the following research question:

> *How can we improve the use of good practices in the agile software development process of a small sized company?*

To answer this question, we did an exploratory case study research with the following elements:

- We used theory on agile methods to describe that the agile software development process consists of the collection of agile methods used.
- We used theory on practices to describe what a good practice is.
- We used 20 dimensions to describe the context of the company's projects, so they can be easily compared.
- We analyzed available documents in the company, conducted interviews with managers and a developer, and observed the actual software development process to determine what the root causes of the problems are the company experiences and what good practices the

company uses.

Based on the data we collected and analyzed, we recommend the following process for improving the agile software development process:

1. Investigate which problems the company has and find out what the root causes of these problems are. We recommend using a problem bundle to visualize the dependencies among these problems.

2. Keep track of the good practices that the company uses and doesn't use. We recommend writing the practices in the form of patterns to keep a consistent structure between practices.

3. Try to match potential useful unused practices to the problems the company experiences and implement these patterns to solve the problems.

4. Have managers and developers share among each other which patterns they are using in which projects, through a knowledge management system, like for example a Wiki. This also allows new employees to quickly gain knowledge how the company works, and ensures that knowledge is kept in the organization if someone leaves.

We believe this process not only leads to an improved use of good practices in the agile software development process of a small company, but also encourages managers and developers to find solutions to problems they are experiencing, while also keeping knowledge about the development process inside the company.

About the title

The title of this thesis is based on two groups of stakeholders in Scrum: the so-called “pigs” and “chickens”. The names of these groups come from an old joke, as explained by Ken Schwaber (2004):

> *A chicken and a pig are walking down the road. The chicken says to the pig, “Do you want to open a restaurant with me?” The pig considers the question and replies, “Yes, I’d like that. What do you want to call the restaurant?” The chicken replies, “Ham and Eggs!” The pig stops, pauses, and replies, “On a second thought, I don’t think I want to open a restaurant with you. I’d be committed, but you’d only be involved.”*

The “pigs” are those stakeholders who have a management responsibility in the project, while the “chickens” are stakeholders that have no management responsibility, but do have an interest in the project.

In other words, the title of this thesis means this research tries to ensure the “pigs” are being committed to the project and are taking their share of the project’s management responsibility. More on the Scrum roles can be found in paragraph 3.1.2.

Preface

This report concludes my research into agile software development for the master Business Information Technology (BIT) at the University of Twente, in Enschede. The actual research for this thesis is done in a small sized company. With this master thesis I finish the BIT curriculum and obtain the Master of Science grade.

The report and it results would not be possible without the help and support from my supervisors from the University of Twente, Maya Daneva, and Chintan Amrit. Their constructive feedback and suggestions always motivated me a lot. Also their knowledge on the subject and relevant literature gave me new directions in which to continue and provided me with new insights on the subject.

Secondly, I would like to thank my colleagues at the company, who provided me with the opportunity and space to complete this research. They were very constructive with providing input throughout the duration of my research, even when I asked them (too) many questions. They never doubted my ability to complete this research.

Thirdly, I would like to thank my family and friends for their continued support.

Last but not least, I would like to thank my girlfriend for her ability to motivate, even when my motivation was low.

Enschede, June 2011

Roel van der Hoorn

Table of contents

List of tables

List of figures

1 Introduction

1.1 Research setting

This research takes place in a small company in Enschede, The Netherlands. The company operates on national as well as international level, delivering web-related IT services and software. Its main activities are web development and web hosting. The structure of the company is flat: a small management team accompanied by 5-10 highly-educated developers. Developers are encouraged to take on various other tasks, besides software development.

In the field of web development, the company aims to deliver high quality software in a flexible way: high quality software in terms of customer satisfaction, a low number of bugs and high maintainability; and flexibility in terms of the ability to adapt to the customer's wishes and needs.

1.2 Research content

During software development requirements tend to change, stakeholders have different priorities, the priorities of the requirements for the business change, stakeholders may not have the required skills to elicit requirements, and developers may interpret requirements differently than the customer does (DeGrace & Stahl, 1990; Alexander & Robertson, 2004). This leads to delivered software not being aligned with the customer's wishes and needs, and ultimately results in a low customer satisfaction. If changed requirements are addressed in subsequent releases, the overall system may become less maintainable, leading to a higher number of bugs. The company is trying to address the customer's needs as best as possible by aligning the software requirements as early as possible in the software development cycle.

1.2.1 Agile software development

To be able to align the software requirements early in the development cycle, the company applies agile development methods. It also uses these methods to adhere to its software development goals

(i.e. high quality software in a flexible way). The agile development strategy applied by the company is a combination of Scrum (Schwaber, 1996) and Extreme Programming (XP) (Beck, 1999), as well as parts of pragmatic programming (Hunt & Thomas, 1999). Both Scrum and XP focus on regular customer feedback to be able to align requirements (Beck, 1999; Schwaber, 2004); XP encourages refactoring and test-driven development (TDD) leading to, respectively, a high maintainable software product and a low number of bugs (Beck, 1999). By having a software product that has a high alignment of requirements and a low number of bugs, the company tries to accomplish a high customer satisfaction.

1.2.2 Problems

The company's customers are mostly external, causing the product owner to be often located outside the office. Although the company is willing to pursue in active customer contact by meeting on-site with the customer or by training customers who have no experience with agile software development (e.g. Scrum, Extreme Programming), the costs associated with moving the development team on-site and training are found to be too high by some customers. Also, while the company prefers time and materials based pricing, its customers prefer fixed pricing. Subsequently several problems were found during initial investigation of the company's development process, by talking to management and developers:

- The customers often demand a fixed price, but also want to add, change and remove requirements during the software development process.
- The management is not always completely satisfied with the overall development process.
- Developers are not always completely satisfied with the overall development process.

These problems are the main reason for this research.

1.3 Research goal

Based on the problems and stakeholders, we can use a Goal-Question-Metric (GQM) template (Basili,

Caldiera, & Rombach, 1994; Van Solingen & Berghout, 1999) to state the following research goal:

> The purpose of this study is to *improve* the *agile software development process* from the point of view of *developers and managers* in the context of *a small web development company with off-site customers* by analyzing *available documents*, doing a *literature review* and conducting *interviews*.

The research questions are defined in chapter 2.

1.4 Project context

The software development process of a project at the company consists of multiple iterations named sprints. The length of the iterations and the number of iterations depends on the required time for the total development process. After an initial version of the product is launched, the software product often enters a maintenance period, although this is not explicitly defined as such. This maintenance period consists of iterations of irregular length and number, depending on when the customer wants new features and how long it takes to develop these features. For this research we consider both the software development process and the maintenance period, as these periods are often not strictly separated in the company. Most problems indicated by the managers and developers are not specific to one period either. According to Andringa (2008), Figure 1 shows the difference between the Scrum process as designed and the process as used by the company in one of its projects.

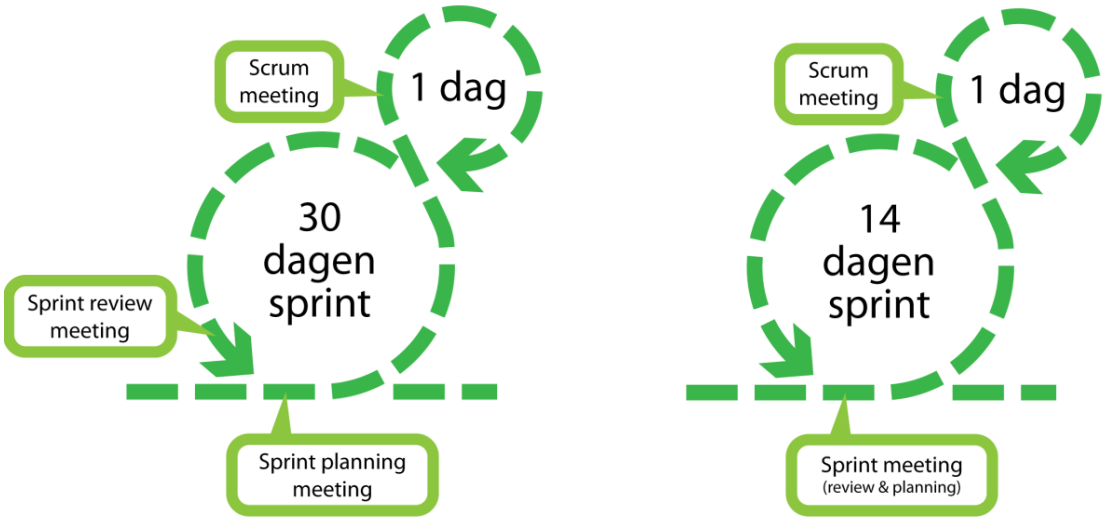

Figure 1: Scrum process and the actual process (Andringa, 2008)

The two process diagrams show that for this project the length of the iterations (sprints) was reduced from 30 days to 14 days, and that the sprint review meeting and sprint planning meeting were combined into one meeting. We leave it out of scope of this research whether 14 days is better than 30 days; Beck and Andres (2004) just mention 'weekly cycle' and 'quarterly cycle' as part of Extreme Programming. More on the context of the company's projects can be found in chapter 4.

1.4.1 Ruby on Rails

The company primary uses the Ruby on Rails framework to develop its web applications. Ruby on Rails (RoR) is a web development framework using the Ruby language that favors convention over configuration. Its main focus is on programmer happiness and sustainable productivity (Hansson, 2009). Thomas and Hansson (2007) describe Ruby on Rails as a framework that makes it easier to develop, deploy, and maintain web applications. The company also uses other Ruby libraries to write its software, as it prefers one language to a set of programming languages; it does also handle integrations between web applications written in Ruby and web applications written in other programming languages. The company also takes over maintenance of existing web applications, as long as they are written in Ruby.

1.4.2 Stakeholders

The software that the company develops, as well as the development process that it uses, is influenced by and has influence on several stakeholders. Alexander and Robertson (2004) apply the onion model to the relationships between stakeholders and come up with a four-layered model. From inner layer to outer layer these are:

- The *kit* or *our product* that is being developed.
- *Our system*, which includes the *product* as well as the people who operate and maintain the product and deliver its results. The *normal operators* deliver these results to the *functional beneficiaries*. *Our system* also includes procedures for operation.
- The *containing system*, which includes *our system* as well as non-operational beneficiaries. *Functional beneficiaries* are people that work in the containing system and benefit from the product. They work for other beneficiaries in the *wider environment*.
- The *wider environment*, which includes the *containing system* as well as all other stakeholders who affect decisions made about *our system*.

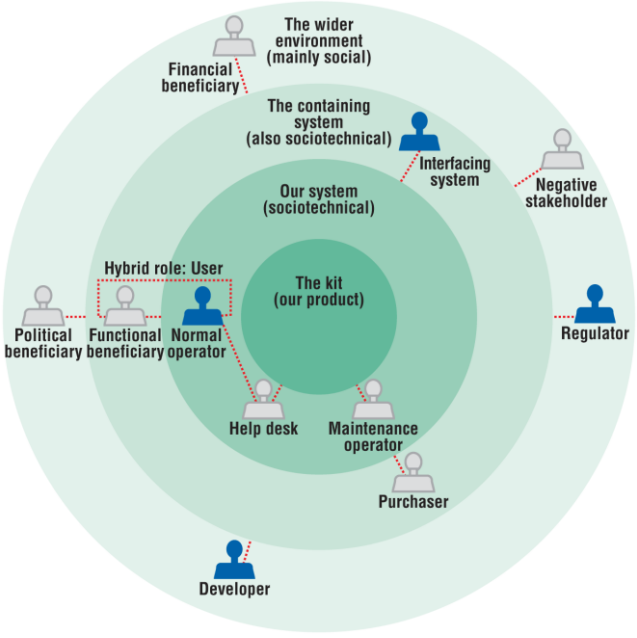

Figure 2: Onion model of stakeholders (Alexander & Robertson, 2004)

The onion model can also be applied to the situation of the company, although the roles are often not explicitly defined as such:

- The *kit* is the web software itself.
- A *normal operator* is usually a person or department of the customer.
- The *helpdesk* is most often located at the customer's site, but it can sometimes include employees, depending on the purpose of the software and who the *functional beneficiaries* are. Often no specific helpdesk is present.
- A *maintenance operator* is usually an employee of the company, but a person or department of the customer may perform some tasks as well.
- A *functional beneficiary* can be a person or department of the customer, a client of the customer or an Internet user, depending on the purpose of the software. The purpose of the software also determines whether the *normal operator* and *functional beneficiary* are the same (i.e. the *user* hybrid role).
- An *interfacing system* is a different (web) application which the software should integrate with. This can be an application owned by the customer or a third party service. The integration can be part of a Service Oriented Architecture (SOA), but not necessarily. Most often the Representational State Transfer (REST) architectural style is used (and preferred by the company), but SOAP and RPC are sometimes used as well.
- The *purchaser* of the system itself is the customer, but the customer may charge its clients for results as well; so in that case the clients of the customer are *purchasers* as well. Again, this role depends on the purpose of the software.
- The *political beneficiary* is the customer's company itself or a director or manager within the customer's company.
- The *financial beneficiary* is the company itself, as it is paid for its development activities. If clients of the customer are *purchasers*, the customer is also a *financial beneficiary*.

- A *developer* is an employee of the company.
- A *negative stakeholder* is, if present, most often located within the customer's company.
- A *regulator* is, if present, often some form of government body.

The list above shows that many stakeholders influence the requirements of the software application. It is therefore important that regular communication within the company takes place, as well as communication between the company and its customers.

1.4.3 Scrum roles

Scrum tries to limit the communication problems of requirements alignment by defining (only) three roles: the *Product Owner* (or customer) who is responsible for the requirements and the release plans, the *Team* who is responsible for developing the product, and the *ScrumMaster* who is responsible for the overall Scrum process. All management responsibilities for a project are divided among these three roles (Schwaber, 2004). More on the Scrum roles can be read in paragraph 3.1.2. According to Andringa (2008), the company implements the three roles as well. More on responsibilities can be found in the interviews in chapter 6.

1.4.4 Extreme Programming roles

Beck (2000) describes several roles when 'working extreme': the *Programmer*, the *Customer*, the *Tester*, the *Tracker*, the *Coach*, the *Consultant* and the *Big Boss*. The Programmer and Customer are the two primary roles (Beck, 2000; Martin, 2000), while the Tester is not a separate person, but only a separate role (Beck, 2000). Beck also writes that if you have people who don't fit the roles, change the roles, don't try to change the people. More on these roles can be read in paragraph 3.2.

2 Research design

Now that we have described the research setting, problem statement, research goal and project context, we can model a proper research design. This chapter identifies what the research questions are, how they are being answered and what phases the research consists of. Verschuren and Doorewaard (1998) describe two groups of activities that together form the basics of designing a research, namely the conceptual design and the (research-)technical design. We will use both the conceptual design and the technical design to structure our research. The conceptual design consists of the research goal, the research model, the research questions and definitions. The technical design consists of the research material, the research strategy and the research planning. As the planning was for internal use only, we do not include it in this thesis.

2.1 Research goal

The research goal, based on a GQM template, was already stated in paragraph 1.3. For completeness it is also stated below:

> The purpose of this study is to *improve* the *agile software development process* from the point of view of *developers and managers* in the context of *a small web development company with off-site customers* by analyzing *available documents*, doing a *literature review* and conducting *interviews*.

2.2 Research model

The research model is a schematic and visual way of displaying the necessary steps for doing research to reach the before-mentioned research goal (Verschuren & Doorewaard, 1998). Figure 3 shows the steps for our research. Each box at the right of an arrow is the result of the steps before. The letters below each column, e.g. *(a)*, denote a phase in the research.

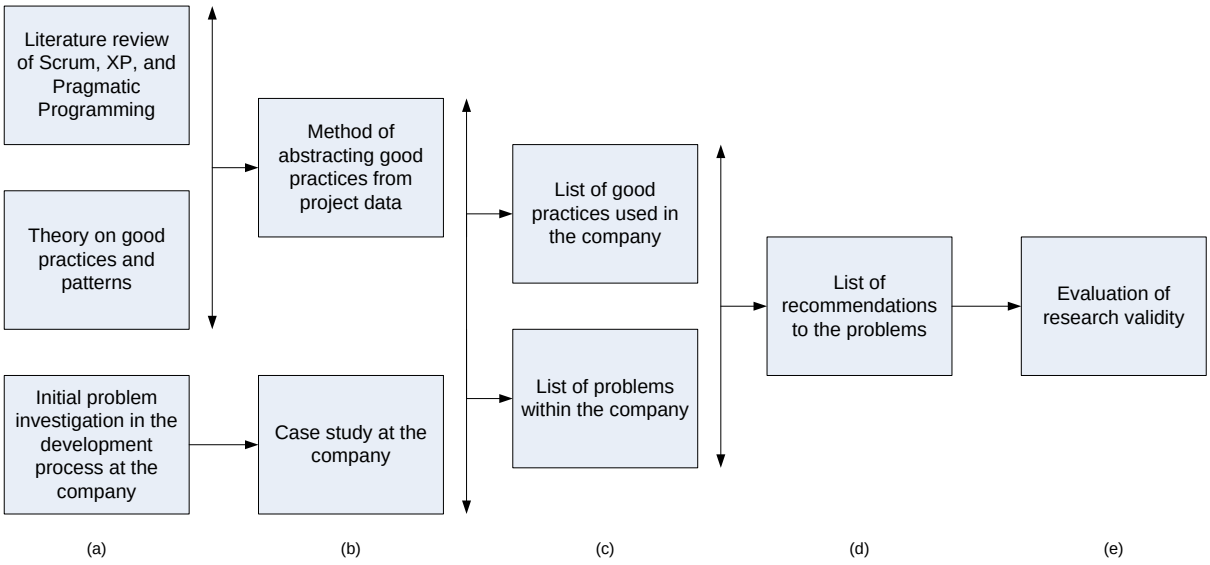

Figure 3: Research model

2.2.1 Phase (a)

In phase (a) we do a literature review of Scrum, Extreme Programming (XP) and Pragmatic Programming, and investigate the theory on good practices and patterns. We use the theory on good practices together with the practices used in Scrum, XP and Pragmatic Programming to create a method for extracting good practices from projects. We also use the initially reported problems in the company (paragraph 1.2.2) to support our case study.

2.2.2 Phase (b)

In phase (b) we analyze the company based on the context of its projects, the documents (and information systems) used, and interviews with relevant stakeholders. Together with the method for abstracting good practices from project data, we create a list of problems that appear in the company and a list of good practices that the company uses. We also add additional literature where appropriate to support our findings and to add external validity.

2.2.3 Phase (c)

In phase (c) we analyze the company's problems and try to find patterns used in agile software development that may provide solutions to these problems. We also try to find good practices that

the company uses and write these in the form of patterns, so they can be used by companies in a similar context.

2.2.4 Phase (d)

In phase (d) we analyze the recommendations (i.e. patterns) to see how they can be implemented in the company.

2.2.5 Phase (e)

In phase (e) we evaluate the validity of our research. In paragraph 2.5.1 we list tactics to increase the validity of our research, which we revisit in our evaluation. We also ask the interviewed people in the case study to value our recommendations to the problems we found.

2.3 Research questions

Based on the research goal (paragraph 2.1) and research model (paragraph 2.2) we deduct our research questions. The following main research question applies:

How can we improve the use of good practices in the agile software development process of a small sized company?

Based on the main research question we can state the following research questions:

1. *How are good practices used in the agile software development process?*
    a. *What do we mean with the agile software development process in this research project?*
    b. *What do we mean with a good practice in this research project?*
    c. *How are good practices used in Scrum, Extreme Programming and Pragmatic Programming?*

2. *How can we describe good practices in a uniform way?*

3. *How can we extract the use of good practices from project data?*

    a. *What methods exist for describing the context of projects?*

    b. *What methods exist for facilitating experience reuse among software managers and/or developers?*

    c. *How can we institutionalize the use of good practices in an organization and promote it among software managers and/or developers?*

2.4 Research material

To let the potential solution be meaningful, we need to gather empirical evidence that supports the existence of the problems mentioned in paragraph 1.2.2. Verschuren and Doorewaard (1998) mention five different types of information sources: people, media, reality, documents, and literature. For our research media and reality are not relevant and thus will not be used.

- **People:** For our research we use interviews with developers and managers. Interviews may result in finding potential problems not easily found using documents. The results of the interviews are discussed in chapter 6.

- **Documents:** For our research we use, among others, a bug tracker, source repositories, a time tracking system, e-mails sent to a private mailing list used for communicating between developers, and Scrum deliverables. Documents are preferred over people, as they provide a more objective perspective. Relevant documents are discussed in chapter 5.

- **Literature:** For our research we use relevant literature to create a theoretical framework that should support the gathering of empirical data. We also use literature to find potential solutions. Relevant literature is discussed in chapter 3.

2.5 Research strategy

Verschuren and Doorewaard (1998) mention five ways to conduct research: a survey, an experiment, a case study, a funded theoretical approach, and office investigation. As we want to get an in depth overview of the problems within the company and how they are related, we prefer a case study research over other methods. A case study research is qualitative in depth way of doing research with a relative low number of research units (Verschuren & Doorewaard, 1998). We now describe in more detail how we conduct the case study research.

2.5.1 Case study research

When doing empirical social research, and thus a case study, there are four tests that determine the quality of such a research: construct validity, internal validity, external validity, and reliability (Yin, 2009, p. 40). Yin (2009, p. 41) also describes several tactics than can be used in case study research to deal with these tests. These tactics are categorized by phase: research design, data collection, data analysis, and composition. We will discuss each of the tactics in the appropriate chapter. Below is an overview of all the tactics categorized by phase showing for which test they help to increase the quality of research. Although the tactics are categorized by phase, this does not necessarily mean each tactic only occurs in that phase (Yin, 2009, pp. 40–41), but it allows us to address each tactic in a more structured way.

Phase of research	Case study tactic	Quality test
Research design Chapter 2 and 3	• Use theory in single-case studies • Use replication logic in multiple-case studies	• External validity • External validity
Data collection Chapter 4, 5 and 6	• Use multiple sources of evidence • Establish chain of evidence • Use case study protocol • Develop case study database	• Construct validity • Construct validity • Reliability • Reliability

Data analysis Chapter 7	• Do pattern matching • Do explanation building • Address rival explanations • Use logic models	• Internal validity • Internal validity • Internal validity • Internal validity
Composition Chapter 6, 8	• Have key informants review draft case study report	• Construct validity

Table 1: Case study tactics to improve research validity (Yin, 2009)

Yin (2009, p. 46) describes four different designs how to conduct a case study, based on two variables: single-case versus multiple-case designs, and holistic versus embedded case studies.

2.5.2 Single-case versus multiple-case designs

One of the rationales for a single case study is when it represents an *extreme* or a *unique* case (Yin, 2009, p. 47). Looking at the available empirical studies on agile software development (Abrahamsson et al., 2003; Dybå & Dingsøyr, 2008), none of the studies mentioned in these review papers cover the use of Scrum and Extreme Programming (XP) together. Also none of the studies mentioned in the review papers deal with an external customer. Although there may be companies that operate in a similar context to the company where we do the research in, we could not find any empirical research on this subject. We therefore consider the use of both Scrum and XP in combination with an external customer, not necessarily a unique case, but definitely an extreme case, which results in the choice for a single-case study research.

2.5.3 Holistic versus embedded case studies

Yin (2009, p. 50) describes the difference between a holistic case study and an embedded case study, whether or not the case study involves more than one unit of analysis.

Using common sense, we have come up with several possible units of analysis: an employee, a (Scrum) development-iteration, a development project, and a company. We do not want to limit ourselves to only improve the way an employee works or to only improve single iterations. Instead

we prefer a broader view, as problems may not be related to employees or iterations at all. A focus on employees or iterations would also be limited by the information available; the development team is located in one office, so many conversations between developers take place in person, and are therefore hard to trace. Choosing a company as unit of analysis would mean we would limit ourselves to the software development process as a whole. From initial investigation we found that the software development process differs somewhat between projects; also most of the initial problems we found in the company, as mentioned in paragraph 1.2.2, are typically associated with a project as a whole. Thus we consider a project the unit of analysis. Looking at the company's portfolio, we think most projects are appropriate for inclusion in the case study.

More on the context of the projects can be found in chapter 4. To improve the internal validity of this project, we will interview people with different perspectives on the project, i.e. managers and developers. To improve external validity, we use literature to see what problems similar companies experienced and how they solved these problems.

2.5.4 Multiple sources of evidence and chain of evidence

To improve construct validity, we use multiple sources of evidence (people, documents, and literature, as explained in paragraph 2.4. We also try to establish a chain of evidence by using a traceability matrix (Appendix B: Problem bundle traceability matrix) to allow readers to find for each problem on which source or sources of evidence it is based.

2.5.5 Case study protocol

We do not explicitly use a case study protocol, as suggested by Yin (2009, p. 79–82). Instead we use the protocol for a research as suggested by Verschuren and Doorewaard (1998), as we are more familiar with its layout.

2.5.6 Case study database

According to Yin (2009, p. 118–120) it is important to develop a case study database with all the

material used in the research. This way other researchers can check the relevant data for more information and can check if no data was omitted in the research. We keep track of all the documents that we used in the research in chapter 5 and we include an overview of the context of all the projects that we investigate in Appendix A: Project dimensions. We record all the interviews that we hold, so that other researchers can listen to them as well, although we do not include the full interviews in text in this research.

2.6 Expected results

The results of this research should be three-fold:

1. A process how the company can improve the use of good practices within the organization. This includes recommendations on how managers and developers can share practices which each other.

2. Some kind of model that allows the company to find the root causes to its problems and to find out which problems are most important to deal with.

3. A table with relevant patterns used (and not used) in the company to be able to find potential solutions to the problems in the model.

We believe the latter two deliverables are important to ensure the process becomes more structured.

3 Relevant literature

Below we will describe relevant literature to the company's case to position our research. As Yin (2009, p. 41) mentions in his case study tactics, it is important in single-case studies to use theory for internal validation.

We first use relevant literature on agile software development methods to understand the characteristics of these methodologies. For this research we limit ourselves to relevant methods for the company, i.e. Scrum (paragraph 3.1), Extreme Programming (XP) (paragraph 3.2) and Pragmatic Programming (paragraph 3.3). Next, we describe the theory on software patterns to understand what patterns are and how they work (paragraph 3.4). We then examine each agile software development methodology on what software patterns they (implicitly or explicitly) use. The results allow us to compare the actual use of agile methodologies in projects to how these are supposed to be used, based on the relevant patterns. In other words, we can compare the patterns used in projects with the patterns that are supposed to be used when using the before-mentioned agile methods.

3.1 Scrum

Although made famous by Ken Schwaber, the roots of Scrum lie within the paper by Takeuchi and Nonaka (1986), who first use the term scrum to compare the product development process to the scrum used in rugby. They describe a holistic method that has six characteristics that can still be found in the Scrum process as described by Schwaber (2004): built-in instability, self-organizing project teams, overlapping development phases, multi-learning, subtle control, and organizational transfer of learning (Takeuchi & Nonaka, 1986).

Schwaber and Sutherland co-developed the Scrum process in the early 1990s to help organizations manage complex development projects (Schwaber, 2004). It was first formally described by Schwaber in 1996 (Schwaber, 1996), although Scrum got more known after he published a book in

2004 (Schwaber, 2004). In his book Schwaber describes Scrum as “a simple process for managing complex projects” and as “a framework and set of practices that keep everything visible”. It is also sometimes described as a “development method” or “an extension pattern language” (Beedle, Devos, Sharon, Schwaber, & Sutherland, 1999). Although not specifically designed for developing software, Beedle et al. (1999) specifically apply Scrum to the software development process. Rising and Janoff (2000) do the same thing and describe it as a “process for incrementally building software in complex environments”. Sutherland (2001) states that “the goal of Scrum is to deliver as much quality software as possible within a series of short time-boxes called sprints, which last about a month.”

Beedle et al. (1999) argue that a repeatable and defined process, such as pursued by those that favor the Capability Maturity Model (CMM) is based on assumptions (i.e. a repeatable/defined problem, repeatable/defined solutions, repeatable/defined developers and a repeatable/defined organizational environment) that have been found wrong in practice. These assumptions assume non-chaotic behavior, although small unknowns in the process can have big influences on the result. Instead of non-chaotic behavior, Scrum assumes chaotic behavior and tries to solve the problems associated with incorrect assumptions (Beedle et al., 1999). It does this by moving control from a central scheduling authority to individual teams, and by shortening the feedback loop between customer and developer, between wish list and implementation, and between investment and return on investment (Schwaber, 2004). Rising & Janoff (2000) describe Scrum as time-boxed incremental development with a twist, appropriate for projects where requirements can’t be defined up front and chaotic conditions are anticipated throughout the product development life cycle.

Schwaber (2004) explains Scrum in several terms: the Scrum (process) skeleton, the roles, the flow (or meetings) and the artifacts. Below each of these will be shortly explained.

3.1.1 Scrum process

Schwaber (2004) describes the Scrum process as iterative and incremental. An overview of the Scrum process by Schwaber (2004) is displayed below in Figure 4.

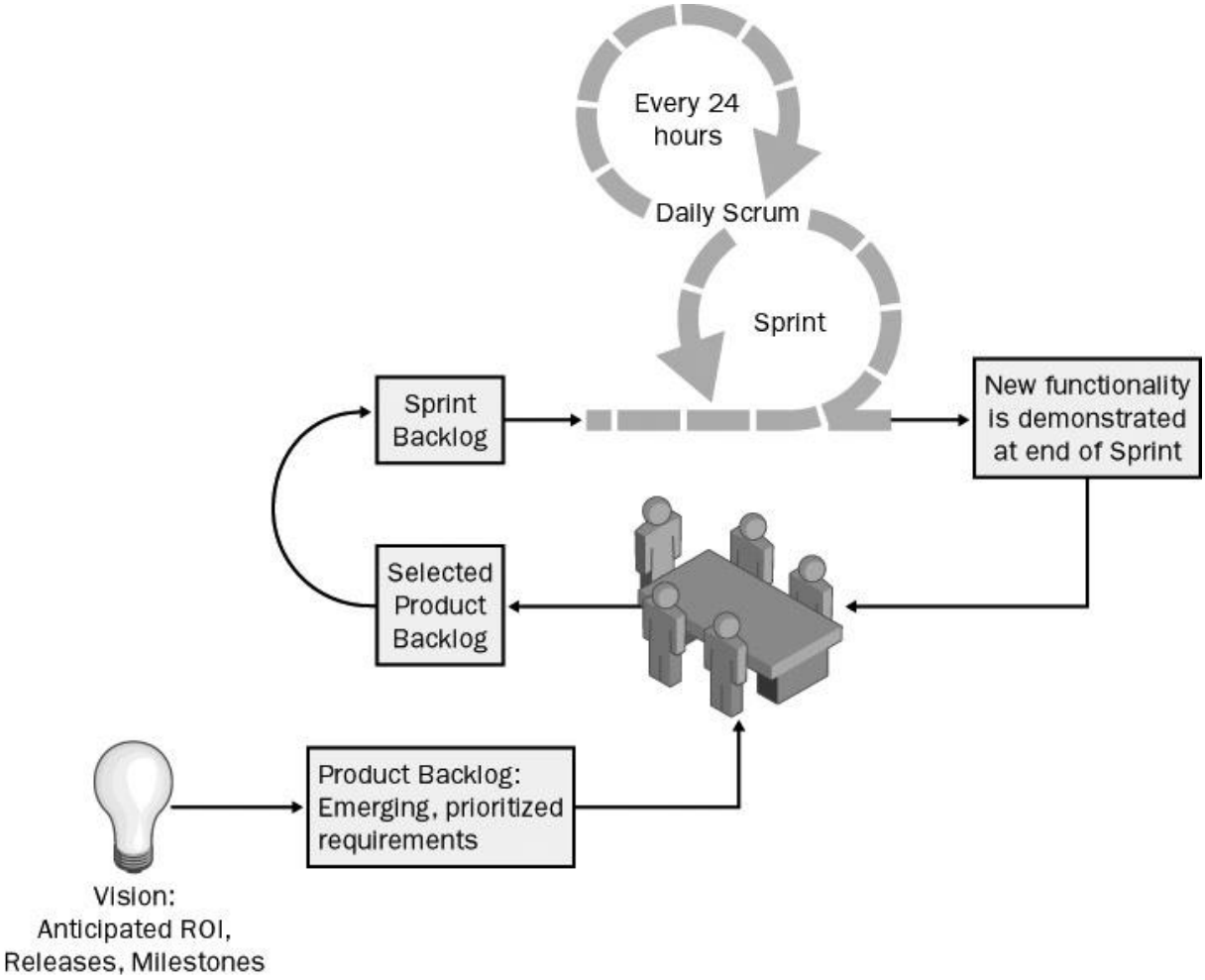

Figure 4: The Scrum process (Schwaber, 2004)

At the heart of Scrum is the iteration, called a *Sprint*. A Sprint is time-boxed, meaning its end date cannot change (Rising & Janoff, 2000). A Sprint typically takes 30 days (Schwaber, 2004). Every day a *Daily Scrum* (paragraph 3.1.3) is held to discuss the status of the Sprint. Tasks that need to be done for the Sprint are placed in the *Sprint Backlog* (paragraph 3.1.4). These tasks are derived from the *Product Backlog* (paragraph 3.1.4), which contains a *list of requirements ordered by priority*. At the end of a Sprint the *implemented functionality is demonstrated*. The Scrum process by Andringa (2008) is displayed below in Figure 5. The *Sprint planning meeting* (paragraph 3.1.3) is held at the

beginning of a Sprint and the *Sprint review meeting* (paragraph 3.1.3) is held at the end of a Sprint.

Figure 5: The Scrum process (Andringa, 2008)

Both figures are not showings the *Sprint retrospective meeting* (paragraph 3.1.3).

3.1.2 Scrum roles

Scrum describes three roles, namely the Product Owner, the ScrumMaster and the Team, over which all management responsibilities are divided (Schwaber, 2004).

- The **Product Owner** is responsible for the interests of those that have a stake in the project and resulting system. The Product Owner should keep track of the Product (i.e. the project's requirements, see paragraph 3.1.4), return on investment (ROI) objectives and release plans. The Product Backlog is kept by the Product Owner to make sure that the most valuable functionality for the stakeholders is produced first (Schwaber, 2004). According to Sutherland (2005), the Product Owner owns the business plan for the product, the functional specification for the product, the Product Backlog, and prioritization of the Product Backlog.

- The **ScrumMaster** is responsible for the Scrum process, i.e. its implementation within the organization, the training of everyone involved with the rules and practices, the adherence

to these rules and practices, and the measurement of progress toward the goal of the development of functionality for the Sprint (Rising & Janoff, 2000; Schwaber, 2004). He ensures that everyone makes progress, records decisions made at meetings and keeps the Scrum meetings short and focused (Rising & Janoff, 2000).

- The **Team** is collectively responsible for developing the functionality. They are self-managing, self-organizing, and cross-functional (i.e. covering multiple disciplines). The Team is also responsible for figuring out how to break up items from the Product Backlog into iterations and how to implement these items (Schwaber, 2004).

Although all management responsibilities are divided over the above three roles, this does not mean the people who fulfill these roles are the only people involved. Other stakeholders may be interested in the project as well, but Scrum ensures the stakeholders that have responsibility also have authority and that stakeholders without responsibility have no authority, within the project (Schwaber, 2004).

More information on the roles used in Extreme Programming can be found in paragraph 3.2.4.

3.1.3 Scrum meetings

Scrum has several meetings that allow the stakeholders with responsibilities to inform each other on the status of the project and what needs to be done to reach completion of the project.

- The **Daily Scrum** is a daily short meeting of ca. 15 minutes by the Team, with the purpose of synchronizing the work of all Team members and team-building. Each member of the Team says what he has done on the project since the last Daily Scrum, what he plans to do on the project until the next Daily Scrum and what problems he found that limit him in meeting the goals of this Sprint and project (Rising & Janoff, 2000; Schwaber, 2004). The Daily Scrum is also attended by remote contributors, making them feel part of the Team and making their

work visible to the other members of the Team (Rising & Janoff, 2000).

- The **Sprint planning meeting** is held at the beginning of each Sprint and cannot take longer than 8 hours. It consists of two parts; each part cannot take longer than 4 hours. In the first part the Product Owner explains the items from the Product Backlog with the highest priority to the Team. The team selects as many items at it thinks it can complete in the coming Sprint. During the second part, the Team plans out the Sprint by splitting the Product Backlog items into tasks that are put into the Sprint Backlog, see paragraph 3.1.4 (Schwaber, 2004).

- The **Sprint review meeting** is held at the end of a Sprint and cannot take longer than 4 hours. In this informal meeting the Team presents the implemented functionality to the Product Owner and any other stakeholder that wants to attend (Schwaber, 2004).

- The **Sprint retrospective meeting** is held between the Sprint review meeting of the previous Sprint and the Sprint planning meeting of the next Sprint and cannot take longer than 3 hours. In this meeting the ScrumMaster and the Team try to find ways to improve the development process for the next Sprint, of course constrained by the Scrum rules and practices (Schwaber, 2004).

Documents we found regarding meetings held in the company can be found in paragraph 5.9.

3.1.4 Scrum artifacts

Scrum describes several artifacts (sometimes called deliverables) that are used during the Scrum process (Schwaber, 2004):

- The **Product Backlog** contains the requirements and initial estimation for these requirements for the product being developed. The Product Owner is responsible for the contents, prioritization, and availability of the Product Backlog. The requirements in the Product Backlog are not fixed, but subject to change, meaning that the company (i.e. the business for

whom the product is developed) can adjust the Product Backlog according to its wishes. The Product Owner should ensure the requirements reflect the wishes of the business; the Team estimates the time necessary for implementing the requirements.

- The **Sprint Backlog** contains the tasks that need to be done for that Sprint. Based on requirements from the Product Backlog, the Team defines the tasks for the Sprint Backlog. A single task should not take more than 4 to 16 hours. If a task takes longer than 4 to 16 hours, it should be split into smaller tasks. Only the Team is allowed to change the Sprint Backlog.

- A **burn down chart** is a graph that shows the amount of work remaining across time and the progress of the Team in reducing this work. A burn down chart can be created for a single Sprint, as well as for a release. The burn down chart can be used to estimate when the remaining work is completed, thus showing if the estimated time is accurate. A simple example of a burn down chart is given below, although we notice that various types exist.

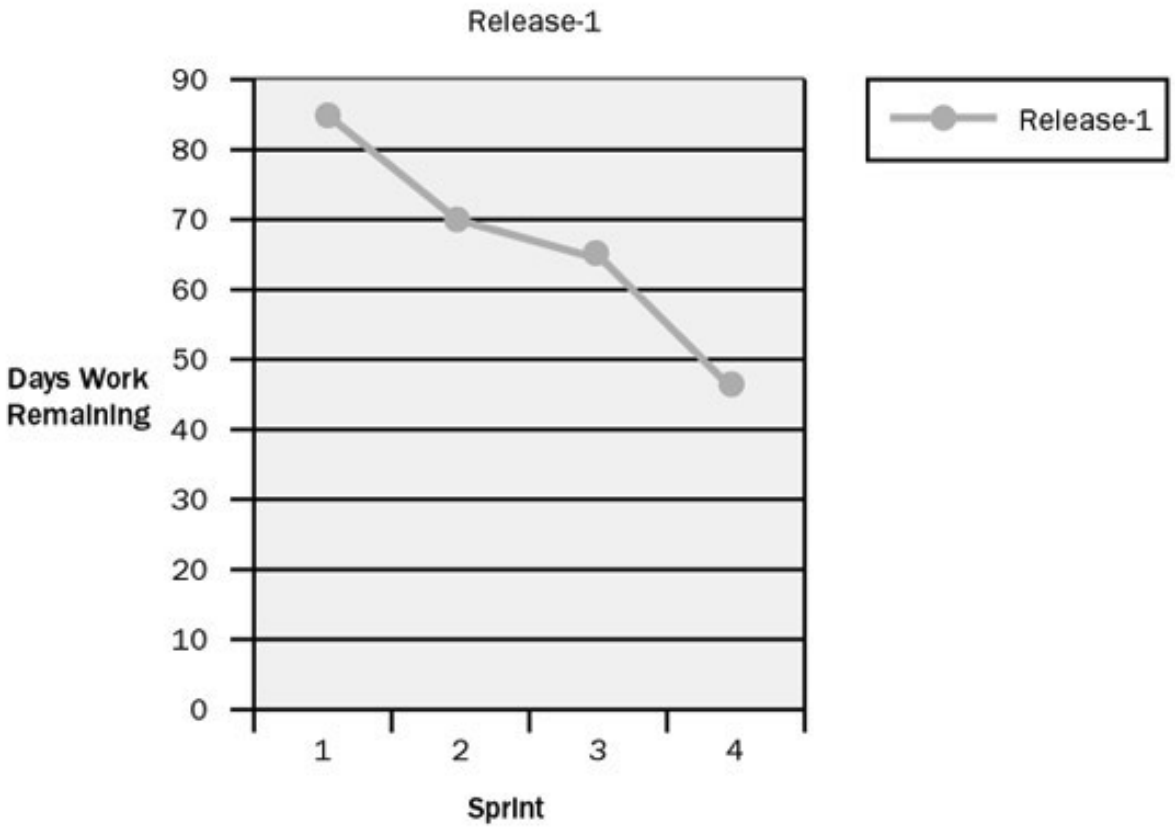

Figure 6: Example burndown chart (Schwaber, 2004)

More information on the artifacts that are used in the company can be found in paragraph 5.8.

3.2 Extreme Programming

Beck (2000) describes Extreme Programming (XP) as a lightweight methodology for small-to-medium-sized teams developing software in the face of vague or rapidly changing requirements." Beck and Fowler (2000) argue that XP "is the programming discipline … we are evolving to address the problems of quickly delivering quality software, and then evolving it to meet changing business needs", explicitly avoiding the word 'method' (or 'methodology'). Wake (2000) calls it a programming discipline too, but also mentions it is a team discipline and a discipline for working with customers. Martin (2000) agrees somewhat by saying the focus is more on people and writes that XP is a "software development method that views people, rather than paper, as a project's most potent element." Don Wells (2002) says that the definition of XP depends on the view: "For some people

Extreme Programming (XP) is a new set of rules, for others it is a humanistic set of values, and to some it is a very dangerous oversimplification." Beck and Andres (2004) state that "XP is a style of software development focusing on excellent application of programming techniques, clear communication, and teamwork" and "XP is a methodology based on addressing constraints in software development". XP includes a philosophy of software development, a body of practices, a set of complementary principles, and a community that shares these values (Beck & Andres, 2004). So while Scrum is not specifically targeted on software development, Extreme Programming (hence the word Programming) is.

Beck and Andres (2004) describe how XP works by defining values, principles and practices:

- **Values** are large scale criteria people use to judge what they see, think and do; they are also universal (Beck & Andres, 2004). Values are ideals; they are abstract, identifiable and distinct (Shore & Warden, 2007). The values that are used in XP are described in paragraph 3.2.1.

- **Principles** are the connection between values and practices; they can be considered domain-specific guidelines (Beck & Andres, 2004). Principles are applications of values to an industry (Shore & Warden, 2007). The principles of XP are described in paragraph 3.2.2.

- **Practices** are evidence of values. Values bring meaning to practices and in turn practices bring accountability to values (Beck & Andres, 2004). Practices are principles applied to a specific type of project (Shore & Warden, 2007). The practices that are used in XP are described in paragraph 0.

While these three levels of knowledge are important, they do not describe XP completely. Beck and Andres (2004) as well as Shore and Warden (2007) describe several **roles** that may be useful to have in the Team, or it could happen that people with a certain role may be involved in the project. However, while Beck and Andres (2004) note that having certain roles can be of benefit to the Team, there is no necessary one-on-one relationship between roles and people and not necessarily every

role is present. Using roles may provide means to create better software, but defining abstract roles is no goal by itself. Roles possibly used in XP are described in paragraph 3.2.4.

Shore and Warden (2007) also give a list of **prerequisites and recommendations** for the Team's environment to successfully use XP in the organization.

3.2.1 Values

Five[1] distinct values can be identified in XP: communication, simplicity, feedback, courage and respect (Beck & Andres, 2004; Shore & Warden, 2007). Beck and Andres (2004) note that these five values are not the only positive values for effective software development; however they are the driving values behind XP. The values are intended to be used together, not apart. The values of XP balance and support each other.

- **Communication** is giving the right people the right information on the right time (Shore & Warden, 2007). It is the most important value when developing software in teams, although it is not all you need (Beck & Andres, 2004). Communicating creates a sense of team and ensures effective cooperation.

- **Simplicity** is discarding things we want, but don't actually need (Shore & Warden, 2007). It makes only sense in context (Beck & Andres, 2004).

- **Feedback** is learning the right lessons at every possible opportunity (Shore & Warden, 2007). As instant perfection is often hard to reach, improvement is the next best thing. It can be achieved by using feedback to get closer to the goals. XP tries to generate as much as the team can handle and as soon as possible by shortening the feedback cycle (Beck & Andres,

[1] In the original book by Beck on Extreme Programming only four values were mentioned (Beck, 1999); a fifth value, Respect, was added in the second edition of the book (Beck & Andres, 2004).

2004). In essence, this is also why there are Sprints and various other regular meetings in Scrum (see paragraph 3.1.3).

- **Courage** is making the right decisions, even when difficult, and to tell stakeholders the truth (Shore & Warden, 2007). That right decision can be taking action, when a problem is known or having patience when the problem is not yet known. However, the consequences of courage should always be taken into account; so the other values are important for 'guidance'. Courage without counterbalancing values is dangerous (Beck & Andres, 2004).

- **Respect** is treating you and others with dignity, and acknowledging expertise and the mutual desire for success (Shore & Warden, 2007). If team members don't care about each other and what they are doing, XP won't work. Also, if team members don't care about the project, it will not bring the wanted result. (Beck & Andres, 2004).

Beck and Andres (2004) also explain the relationships between the values (i.e. how they balance and support each other):

The simpler the system, the more easy it is to communicate about. Improving communication makes early waste reduction possible, which in turn results in a simpler system. Feedback is a critical part of communication. Earlier feedback results also in a simpler system and (like communication) the simpler a system the easier to it is to get feedback. Courage together with the other values is powerful: the courage to tell the truth generates trust and encourages communication, the courage to discard failing solutions and useless features encourages simplicity and the courage to seek real answers generates feedback. Respect lies below the surface of the other four values.

We leave the measurement of these values out of scope, as we believe they are too abstract to be measured effectively. Also, we feel that measuring these values may lead to conclusions that would be too subjective.

3.2.2 Principles

Although values are important to keep in mind, they don't provide concrete advice what to do in software development. In other words: values are too abstract. Principles bridge the gap between values and practices that are in harmony with these values (Beck & Andres, 2004). Beck and Andres (2004) list various principles that guide XP, although these principles are not the only principles used in software development. Other principles may be important, but are not necessary applicable to all software. The principles that guide XP are: humanity, economics, mutual benefit, self-similarity, improvement, diversity, reflection, flow, opportunity, redundancy, failure, quality, baby steps, and accepted responsibility.

We will not explain all the principles in detail here. The principles can be used to understand how to apply a certain practice in the context of a project. In other words: the principles tell what the practices are trying to accomplish. As with values, we consider these principles out of scope for our research, as we feel that they are not defined in a way they can be measured effectively.

3.2.3 Practices

According to Beck and Andres (2004) practices are situation dependent, but there is no fixed list of situated, context-dependent practices that covers all of software development. In their book, Beck and Andres (2004) list 24 different practices. Some of these practices are directly copied from Beck's first book (Beck, 1999); others are renamed or were described implicitly in the first book.

Shore and Warden (2007, p. 25-27) list almost the same practices, but add some, remove some and cross reference them with the first edition of XP (Beck, 1999), the second edition of XP (Beck & Andres, 2004) and Scrum.

For each practice we investigate whether and how it is used in the company. The results can be found in paragraph 7.2.

3.2.4 Roles

Beck and Andres (2004) mention that roles on a mature XP team aren't fixed and rigid, but having them supports everyone in contributing to the best he can. This also means roles and people do not necessarily have to be mapped one-on-one. This is somewhat similar to the cross-functional team in Scrum (paragraph 3.1.2). However, in contrast to Scrum, in XP no explicit distinction is made between roles with responsibility and roles without.

Beck and Andres (2004) mention the following roles: testers, interaction designers, architects, project managers, product managers, executives, technical writers, users and programmers.

In the interviews with relevant people (chapter 6) we provide more insight into the roles used in the company, although we leave it largely out of scope.

3.3 Pragmatic Programming

The Pragmatic Programmer is a book written by Hunt and Thomas (1999) that aims to "cut through the increasing specialization and technicalities of modern software development to examine the core process—taking a requirement and producing working, maintainable code that delights its users." The book covers principles for software developers. Note that these principles are somewhat different than the principles of XP (described in paragraph 3.2.3). The principles of Pragmatic Programming should be considered more an equivalent to the practices of Extreme Programming.

As there are quite many principles mentioned in the book (Hunt & Thomas, 1999), we will not list them here. Also the principles are on a too low level relevant for this research, although we do think that many of the patterns used in Pragmatic Programming are actually in use in the company.

3.4 Organizational patterns

As the company uses a combination of Scrum, Extreme Programming and Pragmatic Programming, our focus will be on those three methodologies. According to Rising and Janoff (2000) Scrum provides empirical controls that enable the development organization to be as close to chaos as the

organization can tolerate. They successfully applied the Scrum process to their organization, saw an increase in productivity, and noticed that Scrum combines and uses organizational patterns that they previously encountered. Beedle et al. (1999) describe the relationships among organizational patterns used in Scrum and other organizational patterns. They also mention that by combining the Scrum patterns with other organizational patterns, it leads to a highly adaptive software development organization.

The building architect Christopher Alexander was the first who described patterns (Alexander, Ishikawa & Silverstein, 1977; Alexander, 1979). Although he used patterns for describing recurring problems and solutions in creating cities, towns, neighborhoods and buildings, it was later applied to any environment that has recurring problems and solutions (Rising & Janoff, 2000). Coplien and Harrison (2004) mention that practices provide a way to combine "broad invariant practices of socially build artifacts" with "specialized practices of individual disciplines" and the relationship between them.

Looking at software, Coplien and Harrison (2004) chose organizational structure (specifically the structure of relationships between roles) as a basis for system understanding, instead of process-based approaches, such as ISO 9000. Each organizational pattern describes a problem that occurs often in the organizational context and then describes the core of the solution to that problem, in such a way that the solution can be different every time you encounter such a problem (Rising & Janoff, 2000). So each pattern solves a problem by adding structure to a system (Coplien & Harrison, 2004). In software engineering recurring problems and solutions have been found on all levels of software development, e.g. from high-level architecture to implementation, testing and deployment (Rising & Janoff, 2000).

In 1996 Kent Beck wrote about best coding practice patterns for Smalltalk (Beck, 1996). Later, in 1999 in 2000 he would use his experience with patterns to write on Extreme Programming (XP)

(Beck, 1999; Beck & Fowler, 2000). Although XP doesn't include a list of patterns, it builds heavily on them (Beck & Andres, 2004; Coplien & Harrison, 2004). Beck (1999) explains that XP is based on best practices for developing software, e.g. unit testing, pair programming, and refactoring. Kerievsky (2000) notes that including patterns into the development organization improves XP, especially the refactoring part.

Coplien and Harrison (2004) use the following form for organizing the important components of a pattern:

Title	The title of the pattern.
Context	The context in which the pattern is applicable.
Problem	A description of the problem that arises in the context.
Forces	A description of the forces that describe the problem.
Solution	A solution to the problem.
Rationale	A rationale why the pattern should be successful.

Table 2: Structure of a pattern (Coplien & Harrison, 2004, p. 22)

Amrit (2008) mentions four types of patterns that deal with software development: design patterns, analysis patterns, organizational patterns, and process patterns.

Although the title of the book by Coplien and Harrison (2004) specifically describes organizational patterns for use in agile software development, it is broader than agile development and more concerned with effective software development, according to the authors.

So, according to Coplien and Harrison (2004), we can use organizational patterns to provide a way to combine human structures with the best practices of software development. To establish what

(organizational) patterns are used by the company, we need a) to find a method to extract patterns from available project data, and b) to understand what makes a practice a best (or good) practice, so we can evaluate the validity of these patterns.

3.5 Extracting patterns from project data

Petter and Vaishnavi (2008) use narratives to facilitate experience reuse among software project managers, which they name Experience Exchange. The software project managers write these narratives and store them in a so called Experience Exchange Library. Other (novice) project managers can use these narratives to improve their understanding of managing projects (Petter, 2008; Petter and Vaishnavi, 2008).

We believe that we can achieve something similar in our case by using a pattern as a substitute for a narrative. Thus developers and managers can write patterns based on their experiences in a project. By storing and sharing these patterns, other developers and managers can use (and potentially refine) these patterns in other projects.

3.6 Validity of good and best practices

Daneva and Ahituv (2010) provide a literature review of the ways in which software engineering practices are judged and the approaches used to evaluate the validity of specific practices. Only in a relatively small portion of the reviewed sources the practices were evaluated on validity. In this portion of the reviewed sources a practice is labeled ‘good’ or ‘best’ if successful organizations use the practice frequently in their projects, based on the assumption that successful projects follow sound engineering principles, while failing projects do not. However, Daneva and Ahituv (2010) observe that for the majority of the practices, there is no statistically representative evidence of the good practice status of these practices. This status is for the majority of the practices exclusively based on anecdotic evidence in successful software projects. Daneva and Ahituv (2010) also note there is no agreement on what a valid practice is or what validity means; quantitative as well as

qualitative research techniques were used. Jones (2000, 2009) explicitly recommends that any potential best practice needs empirical results from at least 10 companies and 50 projects.

To overcome the limitations in determining validity, we include for each potentially new organizational pattern in what projects it is used. Daneva et al. (2007) describe 20 relevant dimensions of projects for the purpose of exploring interface problems of requirements engineering and architectural design. While we feel that the company's problems may not only be related to requirements engineering and architectural design, we still consider the dimensions relevant for this research, as they provide an adequate overview of the company's projects. More on the dimensions can be found in chapter 4.

4 The company's context

To understand the problems the company is experiencing, we need to understand the context in which it is operating. First we show an organization chart of the company. The relationship between the management and the developers is very informal, which results in a relatively flat organizational hierarchy. As the focus of our research is primarily on the software development process, we consider the financial administration out of scope.

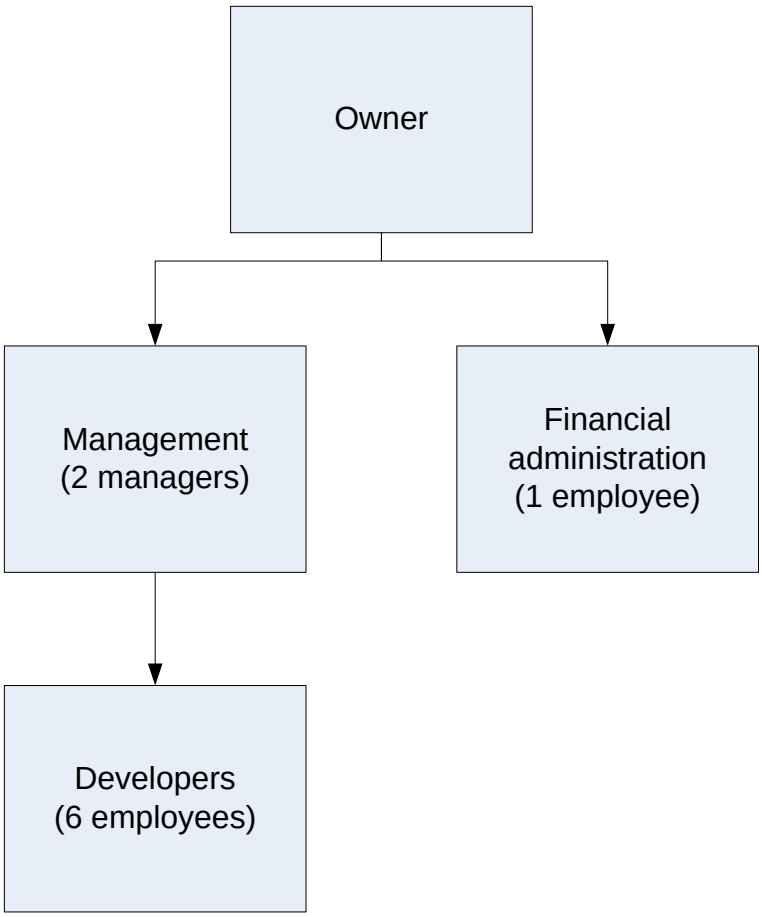

Figure 7: Organization chart of the company

As the company's problems are related to its projects, we need to specify the context of each project to be able to find solutions to the problems, as well as to ensure validity of our research. Daneva et al. (2007) describe 20 relevant dimensions of projects for the purpose of exploring interface

problems of requirements engineering and architectural design. While we feel that the company's problems may not only be related to requirements engineering and architectural design, we still consider the dimensions relevant for this research, as they provide an adequate overview of the company's projects and allow us to compare the projects with each other. We assigned each project a letter, which we use throughout this research.

In paragraph 4.1 we provide an overview of all dimensions by giving the definition of each dimension as written by Daneva et al. (2007). In paragraph 4.2 we provide more information on the actual context of the company's projects. For a detailed overview of the dimensions of each project, see Appendix A: Project dimensions.

4.1 Dimensions

Below we give for each dimension the definition as stated by Daneva et al. (2007). For more information on these dimensions, we recommend the source.

1. **Project nature** refers to whether the project is new or whether it involves some form of modification to existing applications.

2. **Technology gap** addresses how much experience the developers and maintainers have with the technology that the project depends on.

3. **Level of integration** is to reflect the level of integration with other applications. It addresses whether the application is stand-alone or integrated with other applications (and if so, to what extent).

4. **Project organization** addresses whether the project team members are co-located or dispersed at national or international level.

5. **Availability of project resources** refers to whether the project team members are dedicated

resources or shared.

6. **Project relationship structure** addresses 'the business arrangement' behind the project execution. It implies a unique business model, assuming certain roles of the actors in the software industry and certain approach to software funding. It is considered worth keeping a distinction among: in-house projects, contract/subcontract projects, and fixed-price.

7. **Procurement process** is concerned with the type of process used to bid, screen, and select vendors, and acquire the software system.

8. **Market focus** reflects if a project "knows its users by name" at the time of its conception or it is done for mass market.

9. **Business sector** is about the business sector of the organization that uses the application.

10. **Functional area** identifies the functional area in a business being addressed by the project.

11. **Development organization** reflects the size of the organization (not the specific team) responsible for the project delivery.

12. **Size of the client organization** reflects the size of the organization (not the specific team) that will use the delivered system.

13. **Business risk** reflects the strategic importance of a project and, consequently, the damage to business due to project failure.

14. **System risk** is concerned with risks posed on the environment of the system due to system failure.

15. **Risks to the project** is concerned with the types and severities of the various risks posed to the project.

16. **Project size** is about the functional size of the application being developed or modified.

17. **Duration** reflects how long a project is planned for.

18. **Client experience with IT projects** refers to whether the client organization is technology-aware and used to make technology-related decisions (while primarily relying on their own well-informed judgments of their options).

19. **Methodology used** addresses aspects of the methodology a project team adopts. These sub-dimensions are considered important:

    a. how iterative it is

    b. what is the amount of documentation involved

    c. implied notations to be used

20. **Project governance model** addresses some aspects of the approach to managing the project. It covers:

    a. the rules and regulations under which a system delivery project functions with respect to project organization and reporting

    b. the mechanisms put in place to ensure compliance with those rules and regulations

4.2 Context of the projects

In Appendix A: Project dimensions, we provide detailed tables with all the dimensions for each project. The values for the dimensions for the projects are based on a) personal communication with the company's employees, b) information in the company's information systems and various documents (see chapter 4) and c) on informal interviews with the company's employees (see chapter 6). Based on the tables and the information from chapter 4 and 6, we can conclude the following for

each dimension:

1. **Project nature:** We found 11 Greenfield projects, 7 projects that required Enhancement and corrective maintenance and 5 Replacement projects. The projects that required Enhancement and corrective maintenance were all taken over from other development companies or from freelance developers.

   The company's employees (both developers and managers) prefer almost always Greenfield and Replacement projects over projects that require Enhancement and Corrective maintenance. The quality of the source code of projects that are taken over is found a large problem, as well as the lack of (proper) unit and functional tests. From the interviews with the employees, we can conclude that the quality of the source code of projects that are taken over is a *Risk to the project (15)*.

   For future research we recommend looking into applying test driven development to projects that do not have a test suite.

2. **Technology gap:** We found 14 projects without a clear technology gap, 6 projects with a moderate technology gap, and 3 projects with an above average technology gap. Some employees feel there is a higher technology gap when the *Level of integration (3)* with other applications is high: the projects that have a moderate or above average technology gap all have some form of integration. We also think there are higher *Risks to the project (15)* if the technology gap is larger. From the 6 projects with a moderate technology gap, 5 had a medium level of risks to the project, and from the 3 projects with an above average technology gap, 2 had a medium level and 1 a high level of risks to the project. For comparison, from the 14 projects with a non-existent technology gap, 8 had a low level of risks to the project.

   For future research we recommend looking into potential correlations between technology

gaps, level of integration, and risk to the project.

3. **Level of integration:** We found 13 projects with some form of integration and 10 projects that were stand-alone (i.e. no integration at all). See *Technology gap (2)* for information on the relation between the technology gap and level of integration.

4. **Project organization:** The developers within the company are always co-located, while the customer is almost always located elsewhere, except for the 2 projects that were done in-house. The responsibility for project management is almost always shared between the company and the customer.

5. **Availability of project resources:** We found that 16 projects in the company are done based on shared resources (i.e. developers work on multiple projects at the same time); 7 older projects used a combination of dedicated and shared resources.

6. **Project relationship structure:** We found that 18 projects in the company are done as a contractor, 3 are done as subcontractor and 2 projects are done in-house. The projects that are done as subcontractor are all fixed price. Of the 18 projects that are done as a contractor, 4 use time and materials based pricing, the others are fixed price.

   The company's management prefers projects as contractor as it gives them more control and more revenue; management also prefers projects that use time and materials based pricing if the client has a high *experience with IT projects (18)*. The projects that the company did in-house were found to be too complex from the start.

   We recommend that the company takes a careful approach when developing new projects in-house to ensure the new project doesn't become too complex. We also recommend looking more into how users actually use a product instead of finding requirements for all possible cases. More complex situations can be handled in a later release. Thus, in Scrum terms, the Product Owner should be careful in selecting what he really requires to be

implemented.

7. **Procurement process**: The company gets its projects through the channels mentioned below. Although some of these are unknown, we see that the company gets many of its projects through its business network (via-via) and by customers who have previous experience with other projects done by the company. We believe that while these two channels remain important, there is potential in the other channels.

    - Unknown: 7 projects (some of these may be because a client found the company on the Internet, or through one of the channels below)
    - Business network: 6 projects
    - Previous experience: 6 projects
    - In-house: 2 projects
    - Neighborhood: 1 project
    - Tenders: 1 project

8. **Market focus**: As the company's projects are web based, the market focus is often the customer itself on the back-end side of the product, but also mass market on the front-end side of the product. This makes it tough to make a clear distinction on the market focus of each project.

9. **Business sector**: The business sector differs between the company's projects.

10. **Functional area**: The functional area differs a lot between the company's customers.

    For future research we recommend choosing an appropriate classification scheme or nominal scale to describe the functional area, so it is possible to compare projects more

easily.

11. **Development organization:** The size of the development organization (i.e. the company) is always very small.

12. **Size of the client organization:** We found 6 projects with a very small client organization, 8 with a small client organization and 9 with a large client organization. No projects with a medium client organization were found. Like *Market focus (8)*, it is not always easy to determine who the actual client is; for example when the back-end is used by the customer itself, while the front-end may be used by everyone on the Internet, or by a collection of other companies. To overcome these situations we use the size of the client organization of the main user of the product.

13. **Business risk**: We found 9 projects with a low business risk, 12 projects with a medium business risk and 2 projects with a high business risk. The two projects that had a high business risk were supported by venture capital, but had also inadequate budget availability. Based on the company's projects, we believe a higher business risk leads to more involvement by the customer. See also *Client experience with IT projects (18)*.
    For future research we recommend looking into potential correlations between business risk and customer involvement.
    We recommend for the company that if the business risk of the project is low, there is more need for it to be involved in project management, especially regarding management of deadlines.

14. **System risk**: In 16 of the company's projects the system risk is low. In 7 projects the system risk is medium: in all medium cases this is a monetary risk.

15. **Risks to the project**: We identified the risks below for the company's projects. 8 projects had no risk that we could identify; 3 projects had two identifiable risks. Some form of integration

(6 projects) and projects that are taken over (5 projects) are the highest risks according to the company. Only two projects that were taken over are not considered a risk. In one case the project was very small and in the other case the code quality seemed adequate and a test suite was available at the time the project was taken over.

The IT infrastructure of the customer is often a risk when the customer uses caching proxies or outdated web browsers.

For future research we recommend looking into correlations between *Project nature (1)* and *Level of integration (3)* on the one hand and risks to the project on the other hand.

- Some form of integration: 6 projects
- Project was taken over: 5 projects
- IT infrastructure of the customer: 3 projects
- Inadequate budget availability: 2 projects
- Technology that is unknown to the project team: 1 project
- Many different parties involved: 1 project

16. **Project size**: Most projects that the company does can be considered small, only 2 medium and 2 large projects were found. No very large projects were identified.
    - Small: 19 projects
    - Medium: 2 projects
    - Large: 2 projects
17. **Duration**: We found that 8 projects in the company have a duration of less than 3 months, 12 projects have a duration of 3-9 months, and 3 projects have a duration of more than 18

months. We include maintenance in the duration if the project is actively maintained, meaning that some form of service level agreement is present or if the customer regularly asks for new enhancements.

18. **Client experience with IT projects**: We found 9 projects in the company with clients with low experience in IT projects, 5 with medium experience in IT projects, and 9 with high experience in IT projects. We believe that in general clients with a low experience in IT projects require more 'guidance' with managing the IT project. In other words, the company should play an active role in project management if the client does not have that much experience with managing IT projects. On the other hand, the company's management also got the feeling that some clients with a high experience with IT projects also require a more active role by the company on project management. See also *Business risk (13)*.
    We recommend for the company that if the client experience with IT projects is low, there is more need to be involved in project management, especially regarding management of deadlines.

19. **Methodology used:** Overall we see that some form of agile development is used in all of the company's projects. Some projects are more agile than others, but all projects use Extreme Programming (XP) and most projects have a low amount of documentation.

    - In all projects XP was used; in 5 projects this was used in combination with Scrum. One project used waterfall, although user tests were performed during development. One project claimed to use agile, but in practice it seemed more spiral.

    - We found 18 projects with a low amount of documentation, 4 with a medium amount of documentation and 1 with a high amount of documentation.

    - All projects used text in natural language. We found 9 projects that used design mock-ups (although other projects may have used this as well) and 5 projects that

used diagrams (e.g. class diagrams) as documentation.

20. **Project governance model:** We found only 5 projects in the company that used an 'official' project governance model, in all cases some form of agile project governance; in 17 projects we found that in practice some form of agile project governance was used, while it was not explicitly defined for those projects. In one project, we couldn't really make out what project governance model was used.

    We believe that making the project governance model more explicit would result in more awareness by the customer and developers of their responsibilities.

5 Relevant documents

In this chapter we describe the relevant documents we collected for this research. In paragraph 2.4 we wrote that we would gather data from three different types of information types: people, documents, and literature. We described relevant literature in chapter 3. Below we state what relevant documents are used and have been in use in the company. Most of these documents are in the form of information systems. We also interviewed the company's employees and asked them why certain documents and systems are no longer in use and why certain decisions related to these documents and systems were made. The results of these interviews can be found in chapter 6.

5.1 Collecting the documents

To make sure we gather all relevant documents, we first asked managers and developers what systems and documents are used in the development process. To ensure we didn't miss any relevant sources, we used literature to find other potential data sources available in the company, which are used in its projects. Below we first shortly state the relevant documents, after which we describe them in more detail.

From knowledge management perspective (Binney, 2001; Earl, 2001) the following systems are identified within the company: bug trackers (5.2), source repositories (5.3), wikis (5.4) and internal mailing lists (5.5). We also identified a time tracking system (5.6), an invoice system (5.7) and some folders with Word, PDF and Excel documents on the network disk (5.9).

As explained in paragraph 3.1.4 Scrum recommends using the following deliverables (or artifacts): the Product Backlog, Sprint Backlogs, and burn down charts. Although Scrum was explicitly used in only a few projects, we describe if and how these deliverables are used in all projects (5.8).

5.2 Bug trackers

The company has used several bug trackers (also called issue trackers or ticket trackers) in its

lifetime. Most of the time, the bug tracker was managed by the company itself, but in two projects, the bug tracker was initially kept by the client of those projects.

We also discovered that initially the bug tracker was only used internally by the company, but that more recently it's moving its customers to use Redmine as a feedback system as well.

5.2.1 Managed by the company

The first bug tracker used in the development process of the company was Trac. Trac can be described as "an enhanced wiki and issue tracking system for software development projects", while using "a minimalistic approach to web-based software project management" (Edgewall Software, 2010). Trac has Subversion (a type of code repository) support, which was also used by the company at the time.

Later, the company migrated to a different bug tracker named Redmine (Lang, 2010). While the original Trac data is no longer available, fortunately the company moved all its Trac projects to Redmine, including existing data. Thus, no data was lost, although some internal links between tickets did not work well afterwards. Both Trac and Redmine have a repository browser, so developers can browse through the repository.

We found no reason why specifically Redmine was chosen over other bug trackers, although Redmine was created using the Ruby on Rails framework. As the company also uses Ruby on Rails in its projects, there may have been a preference for using a project created with the same technology. We do know no explicit comparison was made between bug trackers, to select the 'best' option for the company.

5.2.2 Managed by others

In two projects a bug tracker was used that was managed by the client.

- In *Project B* a bug tracker named Codebase was used. Codebase also includes repository

hosting, wikis and time tracking (aTech Media, 2010). The reason for choosing Codebase was mainly because the project was taken over by the company, thus the repository and bug tracker were already present.

When *Project B* was taken over by the company, the bug tracker and code repository in Codebase remained in use, while the company used its own wiki for this project.

As the project progressed it was decided, mainly by the management, to start maintaining the tickets in Redmine and move the code to the company's code repository. Tickets in Codebase were not moved to Redmine and Codebase is no longer used in the project, neither by the company or the client. So while Codebase is still active, older tickets are not maintained anymore. As they are all closed, this is not really an issue.

- In *Project X*, a Redmine instance managed by the client was used. As the company's Redmine was not yet set-up to allow customers to create tickets as well, initially the client's Redmine was used. When the company's Redmine was set-up correctly, the project was also moved to the company's Redmine. Tickets in the client's Redmine instance are no longer maintained. As they are all closed, this is not really an issue.

Thus currently for all projects Redmine is used as a bug tracker. The company configured Redmine in such a way that a ticket can currently have one of the following statuses:

- **New**: A new ticket.
- **Agreed:** Either the customer agreed to pay for the estimated time given in the ticket, or it is a bug that the company needs to fix, in which case no time estimation is given.
- **In progress:** Someone is working on the ticket.
- **Resolved:** The company is done working on the ticket. The customer needs to test the result.

- **Closed:** The result of the ticket is accepted by the customer.
- **Feedback:** The ticket requires feedback, most often by the person that created the ticket.
- **Rejected:** The ticket is rejected for whatever reason. Mostly these are duplicate tickets, or the customer filed a bug report that really wasn't a bug report, or the customer is not willing to pay for a certain ticket.

By default in the ticket overview in Redmine, tickets that have either the status Closed or Rejected are not displayed, and are considered closed. Thus tickets that have the status New, Agreed, In progress, Resolved and Feedback are displayed and are considered open.

5.2.3 Management tasks

The company has a dedicated project in Redmine in which managers can keep track of certain management tasks.

5.3 Source repositories

The company uses various source repositories for storing the source of its projects. Sometimes, the source repository is managed by the company itself; sometimes it's managed by the client. Two types of repositories are used: Subversion and Git.

5.3.1 Managed by the company

The first type of code repository used in the development process of the company was Subversion (sometimes abbreviated as SVN). Subversion describes itself as "a full-featured version control system originally designed to be a better CVS" (The Apache Software Foundation, 2010).

There were several problems with Subversion identified by the company, mostly related to managing different versions of the same code:

1. A repository could be forked (branched), but it was not evident to merge changes in the fork

back into the main repository.

2. A single commit made to a branch could not be easily picked out to be applied to other branches of the same repository.

Later, the company migrated to a different source repository named Git. Git advertises itself as “a free & open source, distributed version control system designed to handle everything from small to very large projects with speed and efficiency” (Baudis, P. & Chacon, S.). Only active projects were moved to Git, while older projects were left on Subversion. The Subversion repository is still active, so if code from older projects is required, Subversion can still be accessed. Newer projects are only stored in Git.

5.3.2 Managed by others

Some clients prefer to keep the source code for their projects in-house and some clients have multiple projects, which are not all maintained by the company. In these cases the repositories in use are either Subversion or Git as well.

5.4 Wikis

The company has used several wikis in its lifetime to store knowledge information. In all cases the wiki was managed by the company itself.

5.4.1 Trac

At first, the company used the wiki that was integrated into Trac. As there were Trac instances for each project, the information in each Trac-wiki was mostly project related: e.g. hosting information, deployment information, repository information, and contacts (phone numbers, etc.).

5.4.2 DokuWiki

After the move to Redmine, a new separate wiki was used, called DokuWiki. DokuWiki is described as “a standards compliant, simple to use Wiki, mainly aimed at creating documentation of any kind. It is

targeted at developer teams, workgroups and small companies" (Gohr, 2010). As the company also added hosting to its services, server related information (e.g. how to install, software used, and general layout) was also added to the wiki, besides the project related documentation.

Also some good code practices in Ruby on Rails, plug-ins to use in Ruby on Rails projects and software patches were stored in the wiki. So it appears that during this time a) some tacit knowledge was made explicit and b) some explicit knowledge was centralized. However, updates rarely occurred on the plug-ins to use and good code practices. When updates did take place, it was mostly related to projects and server information, if necessary (e.g. when a username/password changed, or when some more contacts were known), thus the knowledge on what plug-ins to use and good code practices lost its value over time.

We believe the most important reason for not updating the explicit knowledge is the speed in which the explicit knowledge becomes outdated. While some good code practices still remained good practices after a while, most patches and plug-ins to use became outdated fairly quickly:

- Plug-ins were often not well-maintained, so what was "the" plug-in to use at one moment in time, was already an outdated plug-in three months later.
- Patches were most of the time specific patches for Ruby on Rails versions. As there was a new version released every few months or so, the patches quickly lost their value.
- The good code practices were sometimes useful from a higher point of view (i.e. how to solve a specific problem), but the code itself was often targeted at a specific version of the Ruby on Rails framework thus not always directly useable.

So as the speed to which knowledge becomes outdated is relatively high, programmers in the company often rely more on Google and their colleagues, than on the wiki.

5.4.3 Redmine

After DokuWiki had been in use several years, the company moved away from it and back to a more project-centered approach. Thus the still relevant knowledge in DokuWiki was moved to Redmine, which also has wiki functionality. The knowledge is now again stored separate for each project. For specific tasks related to hosting, a separate project is present with associated wiki.

5.5 Internal mailing lists

The company has several internal mailing lists for communication among developers. Some of these are topic based, while others are project based:

- A general office mailing list, which is used for all topics potentially interesting for all developers.
- An admin mailing list, which is used for all hosting and server related topics.
- An exception mailing list, to which all exceptions (page views with an HTTP 500 status code) are emailed by the web server; i.e. when a web application maintained by the company produces an HTTP 500 status code, the whole request, including response is e-mailed. This allows the company to respond quickly to potential bugs.
- Previously there were separate mailing lists for each project to which updates to tickets in Redmine were emailed. Also the exceptions (see previous bullet point) were sent to these mailing lists. Nowadays these mailing lists fell largely into disuse. Updates to tickets are now emailed separately to each developer. Other topics that were previously discussed on these separate mailing lists are now discussed in regular e-mails between developers, managers, and customers. The reasons for moving away from a separate mailing list for each project are that projects are often too small (i.e. a low volume and only a few developers) and that it becomes rather time consuming to manage (e.g. what developer is on what mailing list).

5.6 Time tracking system

As some of the developers now are and in the past were part-time employed, it is important to track the time an employee has worked. Also, the management wants to know which projects run well and which do not. Time tracking helps determining how much time was spent on a project. The company has used several ways for tracking time in its past.

5.6.1 E-mail

In the beginning e-mail was used for communicating the hours worked. Developers e-mailed the hours they worked to the management, which used these e-mails to know how much to pay each developer. These e-mails were sometimes only stating the hours worked on a day, but later by request of the management, became more specific and also stated how much time was worked on specific tasks.

5.6.2 Sherlock

As the company wanted to create products they could sell multiple times to customers, instead of waiting for a customer to ask for a product, they chose a product they could use themselves as well, i.e. a time tracking system. Also the management didn't want to count hours for each employee by hand and to be able to compare indicated hours for a task with actual hours spent.

A system was specified that could, among others, track time, manage companies, manage projects and manage quotes; it was named Sherlock. Although it could do a lot of things, the thing it didn't do well was track time. Adding hours was a rather time consuming process for employees and hours could be added from three different perspectives (general, employee and project). Especially the latter was a real problem as sometimes the perspective changed if the employee added hours. Also the management information gained from Sherlock was not really the information the management needed.

We believe the most important reason for the failure of the project was the lack of proper focus from the start. The system was designed from the start to do a lot of things, but that also made it overly

complex. Although the development team worked in iterations and Scrum was used to guide the development, the system wasn't used until it was complete. So instead of creating a time tracker that worked well in practice and then adding new features, the system had a lot of features, but didn't properly track time in practice.

5.6.3 Back to e-mail

As the employees didn't really track time with Sherlock, it was decided to move back to e-mail.

5.6.4 Excel

As the management still wanted an easy way to be able to count hours worked by their employees, an Excel sheet was created in which the employees could track their time. Every week the employees had to send a filled out Excel sheet to the management.

5.6.5 SlimTimer

To accommodate the need by the management to gain easier insight into which projects run well and which do not, time tracking is now done with SlimTimer. SlimTimer is about "making time tracking easier by eliminating timesheets in favor of using a web based timer" (White, 2006). So SlimTimer does not only allow adding hours to specific tasks, but also allows an employee to track time by starting and stopping a timer.

SlimTimer also provides basic management information about how much hours were spent on tasks and it is no longer necessary to combine data from Excel sheets.

5.7 Invoice system

To manage its outgoing invoices, the company uses MoneyBird. MoneyBird is a web application that allows one to create and manage invoices and quotes (BlueTools, 2010).

5.8 Scrum artifacts

The company explicitly used Scrum in some projects. Below we describe for each Scrum artifact if

and how it is used, for projects in which Scrum was explicitly used but also for the projects in which it was not.

5.8.1 Product Backlog

Only in *Project D* the company explicitly kept a Product Backlog. This project was also the largest project that the company developed. Although Scrum prescribes the Product Owner (i.e. the customer) should keep the Product Backlog, in fact it was the ScrumMaster (an employee of the company) who kept it. We do not know whether this caused any direct problems, but the reason the Product Owner should keep the Product Backlog is that the Product Owner also makes the decision which feature should be implemented next, i.e. the one that makes the decision should also bear the responsibility.

It is possible that a Product Backlog was also kept in other projects, but in that case the customer did not explicitly mention it.

5.8.2 Sprint backlog

There are no projects in which a Sprint Backlog was explicitly kept in document form, but in all projects a bug tracker was used. In almost all projects there were also milestones set in the bug tracker, sometimes even with explicit names like “Sprint 1”. As mentioned in paragraph 3.1.4 the Sprint Backlog contains the tasks that need to be done for that Sprint. Single tasks should not take more than 4 to 16 hours and only the Team is allowed to change the Sprint Backlog. We believe that in most projects the bug tracker adheres to these principles.

Based on this information, we find that the bug tracker can be considered a Sprint Backlog. Still, in some projects, the bug tracker is also used as a feedback system for the customer, in which the customer can add bugs and features. To some extend this allows the developers and customers to communicate directly, but we believe that using the bug tracker as a feedback system should have some rules or guidelines to ensure the customers do not restrict the Team in any way to organize

themselves.

5.8.3 Burn down chart

We couldn´t find any use of burn down charts in the company's projects. In the project in which the Product Backlog was explicitly kept (see paragraph 5.8.1), the Product Backlog did include a "burn down percentage", but no real charts were used. And although the burn down percentage is stated in the Product Backlog, it gives a percentage that only covers the last sprint (i.e. what percentage of the last sprint is done). We believe some form of improvement is possible here.

5.9 Documents on the network disk

Besides the recommended deliverables in the Scrum methodology, some other documents were identified:

- Project proposals, sometimes for tenders. There are also designs (layouts) for some projects. Many projects also have some form of class diagram.
- In the project where the Project Backlog is explicitly kept, there are also documents related to the sprint meetings: agendas, minutes, documents that describe the goals for the next sprint, time indications for potential features, and documents that are used to inform the customer what was done during the last sprint.
- For *Project D* there are also some minutes from sprint retrospective meetings and a log with lessons learned. It seems the log was created in the beginning, but not kept up-to-date, as only three "lessons" are present.
- Some (old) plug-ins and libraries, including often used icons.
- E-books and PowerPoint slides on various topics. Some of the slides were from conferences that were attended by the company's employees.

- Various other administrative documents.

5.10 Conclusions

From the use of documents and systems in the company, we can already conclude the following:

- In *Project D* it seems Scrum was used with most meetings, roles and artifacts present. The actual development process used by the company in this project was shortly described by the ScrumMaster in his bachelor thesis (Andringa, 2008). This thesis also provides us with the reasons for using Scrum within that specific project. More on these reasons can be found in chapter 7.

- It seems the company is moving towards more centralized systems. We have seen that Redmine is now used as a bug tracker, code repository browser, customer feedback system and wiki. We also see that instead of creating mailing lists for each project, there are now only a few mailing lists, unrelated to specific projects. Specific project information is now emailed by developers between each other.

- We haven't seen any burn down charts. Although we haven't discovered specific problems related to the absence of these charts, we think that using these charts (potentially automatically created in Redmine) could improve insight into the progress of the various projects.

6 Interviews with relevant people

In this chapter we describe how the employees of the company work. In paragraph 2.4 we wrote that we would gather data from three different types of information types: people, documents, and literature. We covered relevant literature in chapter 3 and relevant documents in chapter 5. Below we describe how the employees in the company interact with each other, with customers, and with the available information systems and documents. These descriptions are based on informal interviews we held with three employees (two managers and one developer).

6.1 Collecting the information

To collect information on how the employees work, we held informal interviews with three of them (one developer: *Developer A* and two managers: *Manager B* and *Manager C*), specifically asking them how they work, what tasks they usually have, with which documents and information systems they interact and how they interact with them, and what common problems they encounter in the projects they work on.

To improve construct validity (Yin, 2009), we each showed them the results of their interview and asked them for feedback. We also improved the questions in later interviews by using the results from earlier interviews. The interviews were recorded (audio) to ensure we didn't lose any information in the collection process. In the results below, we assigned each interviewed employee a letter, so we know who said what, while at the same time ensuring some form of anonymity. Interviews took between 45 minutes and 1 hour 50 minutes and were held in Dutch. The information in the interviews was translated into English and restructured, meaning that the order of information in the next chapters was not necessarily the order in which the information was received.

To visualize the information from the interviews we add a flowchart diagram and data flow diagram for the results of each interviewee. Next, we combine the individual diagrams of each employee into one flowchart and one data flow diagram, resulting in an overview of the software development

process in the company. For the sake of clarity we separate the management role from the developer role, so one employee can fulfill multiple roles; this allows us to combine the flowchart diagrams more easily.

6.1.1 Legends

The legends below describe the diagrams we use in the following paragraphs.

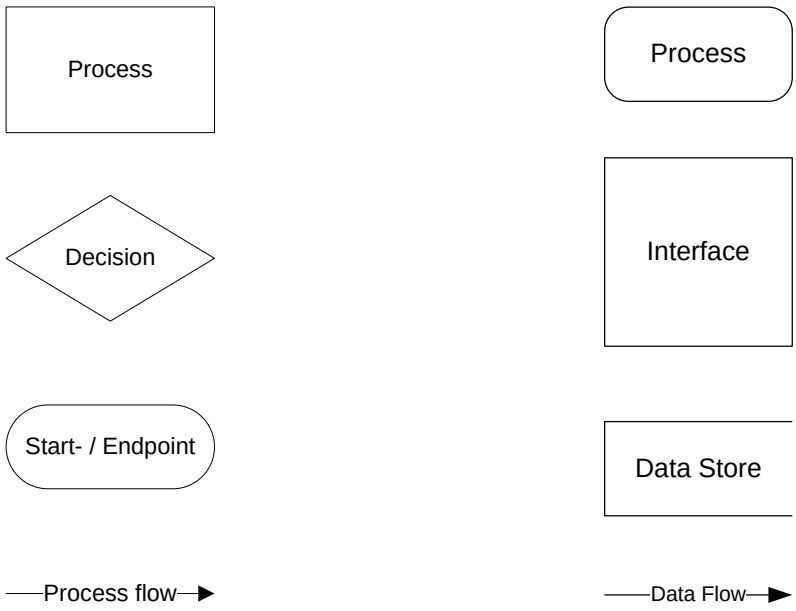

Figure 8: Legends for Flowchart diagrams (left) and Dataflow diagrams (right)

6.2 Developer A

Developer A has worked approximately 2.5 years for the company. In the interview he described the software development process roughly as displayed in Figure 9 below. Paragraph 6.2.1–6.2.6 (also mentioned in Figure 9) explain the development process in more detail. Paragraph 6.2.7–6.2.11 mention other aspects of the development process, not covered by the flowchart in Figure 9.

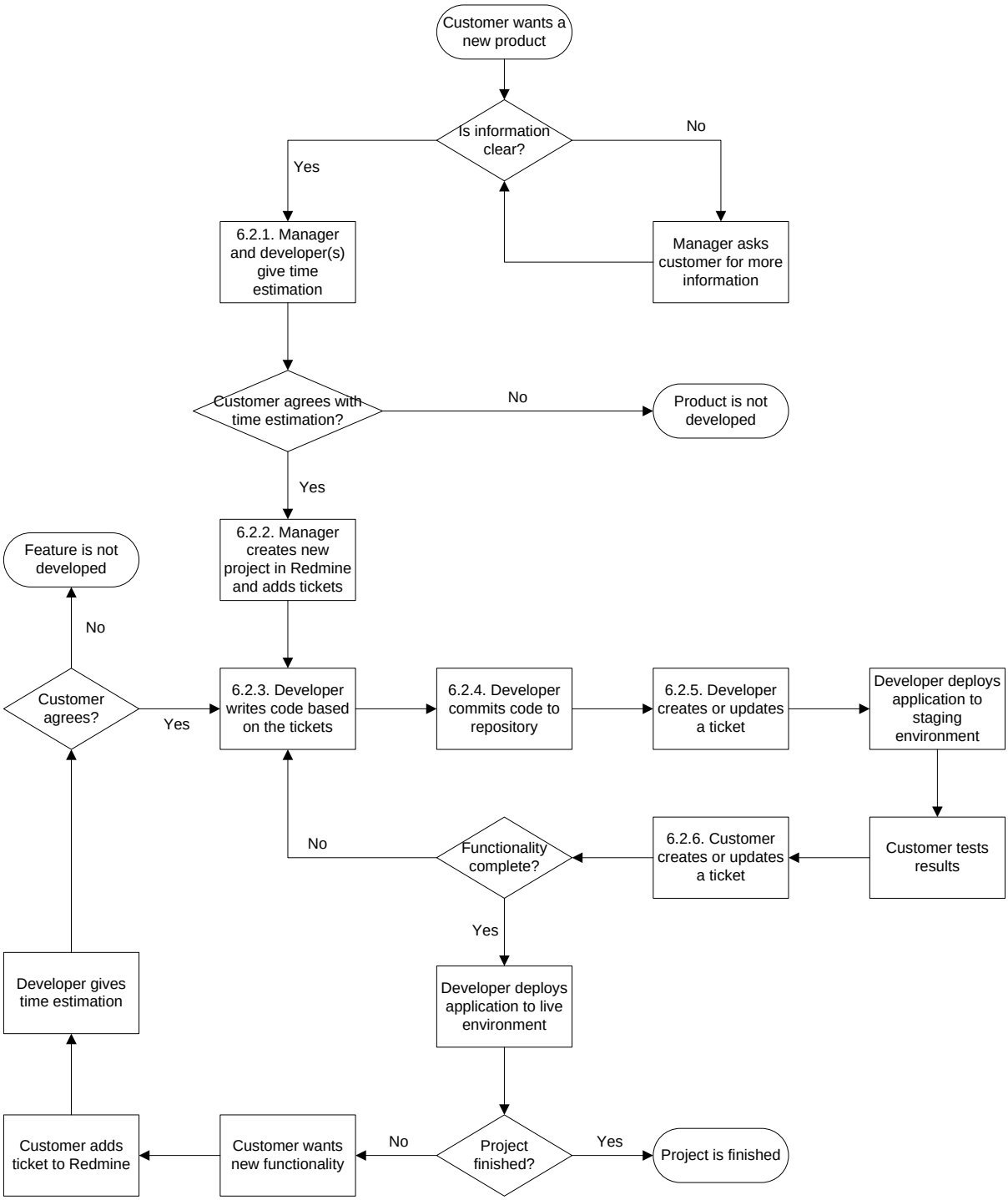

Figure 9: The company's software development process according to Developer A

6.2.1 Manager and developer(s) give time estimation

Depending on the size of the project, a manager and one or two developers together estimate the time necessary to implement the requested features. The larger the project, the more developers are

included. Mostly the features are defined very high-level, which are also mentioned in the quote. During development of the project, the customer can still decide to add or skip certain features.

6.2.2 Manager creates new project in Redmine and adds tickets

One of the problems *Developer A* experiences is that tickets are not always clear enough. According to *Developer A* there are two main reasons why tickets are not clear enough:

- A customer experiences a bug and describes it very briefly or unclearly (see also paragraph 6.2.6). As bugs can often only be fixed if they can be reproduced (Hooimeijer & Weimer, 2007), *Developer A* often needs more information.

- Global backlog items (for example stated in a quote) are directly copied to a new ticket by management, often lacking more specific information, although *Developer A* thinks management is in this case missing information as well. This is mostly due to functionality specified in quotes being (too) global, and thus often unavoidable. On the other hand, *Developer A* thinks its management's task to ensure there is enough information if it creates a ticket.

In both cases the customer needs to be contacted (either by updating the ticket or by e-mail or telephone) to gain more information. *Developer A* doesn't start coding, until he knows enough information, so he doesn't waste time writing something that has a large risk of not being the right feature.

6.2.3 Developer writes code based on the tickets

If *Developer A* experiences a problem during coding to which he has no direct solution, he will ask a colleague if he thinks the colleague may know the answer, otherwise he will use Google. *Developer A* almost never uses the wiki for solving his problems, as he found searching in it not really easy, the information was often outdated and he felt that the Internet is the largest knowledge database, thus it would be illogical to search in a subset (i.e. the wiki). *Developer A* only uses the wiki for project

specific account information (login information for servers, etc.), so he prefers the wiki in Redmine over Dokuwiki. The only other time *Developer A* used the wiki, was when a former employee of the company wrote an article on Ruby blocks (a certain aspect of the Ruby language) and put in the wiki. This article, in contrast with other information in the wiki, remains quite up-to-date.

Developer A almost never uses e-books or conference slides for information. He only used an e-book, called *Agile Web Development with Rails*, when he joined the company. The company has the policy that every new developer should first work his way through this e-book. The e-book is in the form of a tutorial guiding the inexperienced Rails developer through all aspects of the Rails framework in an agile setting. To keep himself up-to-date with the latest news, *Developer A* often watches 'Railscasts' (Railscasts, 2010); sometimes he reads a blog or article.

Developer A actively uses the source repository browser in Redmine. Especially when other people commit code, he wants to know what has changed. The repository browser is also used when a bug report comes in, to see what changed in the past.

6.2.4 Developer commits code to the repository

If *Developer A* wants to commit updated code to a project, he tries to commit the new code to the source repository. If it fails, it is usually because someone else committed code before he did.

Developer A worked in the company with Subversion as well as Git, but prefers Git over Subversion. He finds it easier in Git to make branches (forks), to switch between branches and to merge changes from one branch to another. Therefore he finds Git a large improvement over Subversion. He also feels Subversion generated more conflicts when trying to merge changes from one branch to another. The branching and merging improvement helped, especially in *Project D*.

He finds the only disadvantage is the steep learning curve Git has in comparison to Subversion. Git can do a lot more, but that also means it's more complex to learn and thus more easy to do

something wrong. Still, he thinks Subversion has no real advantage, feature wise, over Git.

Developer A says the company moved from Subversion to Git, because one former employee was very positive about Git. Also Ruby on Rails (and gems and plug-ins often used in projects) moved to Git as well, but he is not really sure whether this was a real reason for the move.

6.2.5 Developer creates or updates a ticket

If *Developer A* needs to create a ticket himself, it depends on for whom the ticket is meant if information is added. He will try to use less technical terms if the ticket is for a customer, than if the ticket is meant for another developer or himself.

6.2.6 A customer creates or updates a ticket

When a customer or other employee (see also paragraph 6.2.2) creates or updates a ticket in Redmine, an e-mail is sent to all managers and developers. It depends on whether *Developer A* is active on the project whether he takes action, but he also actively follows changes to tickets on projects that he worked on in the past.

If so, *Developer A* tries to have an active response to tickets to get the information clear as quickly as possible and to give the customer the feeling that he is taken seriously. He likes it that the customer can directly respond to tickets, instead of sending e-mails or calling to the office. *Developer A* also tries to keep his hands off contact with the customer, besides contact through Redmine. He also feels that Redmine works well if the customer knows how to work with it, but if the customer doesn't communication becomes more cumbersome.

If the ticket is not clear enough, *Developer A* will ask for more information, otherwise it depends on the type of ticket and project how he responds. If the ticket is a bug that is caused by an employee, he will try to fix it, or leaves it to a colleague. If the ticket is a task, suggestion or feature request, he will estimate the time necessary to implement the request if the project requires it (most often it

depends whether the project is fixed price or hourly based).

In one project *Developer A* uses a requirements document (in Dutch *Programma van Eisen*, *PvE*), which was created by the customer before they contacted the company, although the presence of such a document is not common. *Developer A* thinks it's nice to have, as some corner cases are well thought of (especially business logic), but only because it's there. He wouldn't choose to write such a document first, if it weren't available. During development some requirements change, but not much.

Developer A does not have a clear preference for Redmine or previously used Trac, but does think Redmine has a somewhat higher usability. He is in general very satisfied with Redmine, but finds the workflow in Redmine not always evident:

- In the index multiple tickets can be selected and fields can be edited for those selected tickets at once, but other fields cannot be edited in the same way, although you would expect so.
- If a ticket has the status Resolved or Feedback, the ticket is still considered open, but it requires no action by a developer. Instead it is the customer that should act. We believe this limitation of Redmine can largely be explained by the fact that is not meant as a customer feedback system. Still, there are filter options to display only tickets with a certain status.

Developer A also thinks there is more functionality in Redmine than he really uses, but this is not necessarily a bad thing.

6.2.7 An e-mail is received

Besides the obvious trigger, i.e. the customer, *Developer A* is also triggered by different types of e-mails he receives; this e-mail can be by a colleague (manager or developer), a customer, the ticket tracker (Redmine) or a web server. Considering the latter two:

- When a ticket in Redmine is created or updated, it automatically sends an e-mail to all managers, developers and customer contacts that have access to the project the ticket belongs to.

- The company configured its projects in such a way that when the web server running a project generates an exception (see paragraph 6.2.8), it sends automatically an e-mail to the company's employees with details on the exception. The actual visitor of the website gets an HTTP 500 error page if an exception occurs.

Developer A prefers a mailing list per project, although it depends a bit on the project: on a small project there is less need for a separate mailing list, especially when there is only one developer. Currently, *Developer A* has the feeling that he sometimes misses information, because people e-mail each other, but do not necessarily include all people involved in the project. So for workflow *Developer A* finds it nicer when there is always a mailing list for each project.

6.2.8 An exception is received by e-mail

If the exception is found to be critical, *Developer A* immediately takes action. If the exception is not really critical, it depends on the project whether he takes action or not. If the project is a project that he works or worked on, he is more likely to take action. If *Developer A* sees the exception and knows it can be fixed easily, he is more likely to fix it without creating a ticket. If the exception is more serious and there is no obvious solution, a ticket is created with the contents of the exception.

6.2.9 Developer needs to track time

Developer A thinks SlimTimer works well and prefers it over previously used Excel sheets. Using the Excel sheets became one large mess (there was one Excel sheet per employee per week). *Developer A* uses SlimTimer in two ways:

- He starts the timer, so time is automatically tracked.

- He enters the time he worked at a later moment.

Developer A feels there is no real need to integrate SlimTimer or some other form of time tracking in Redmine, although he believes management may like it. He thinks that for management it may still be not clear how time is spent.

Developer A says that the reason for Sherlock falling into disuse is that the workflow was too difficult for adding hours. He also believes that the backend for the management was not easy enough and that it was quite cumbersome to get the necessary results. He thinks the company could have fixed Sherlock, although he didn't work on the Sherlock project himself.

6.2.10 Developer needs functionality that may be present in a gem or plug-in

Developer A likes not inventing the wheel again, so he prefers to use (Ruby) gems or plug-ins if that makes developing easier. The difference between gems and plug-ins is that gems are installed separately from the project and can be included in all project that run on the same computer, while plug-ins are typically project specific. Also, gems always have a version, while plug-ins often do not.

Developer A finds that some experience with gems and plug-ins is useful to avoid certain gems and plug-ins that have caused problems in the past. Also gems and plug-ins that not have received updates in a while usually have to be avoided. He also finds that colleagues often have knowledge about which gems and plug-ins to use and which to avoid.

Developer A finds that if the gem or plug-in has too much functionality for the problem he wants to solve, he prefers to write code himself. If code in the gem needs to be adapted, he does one of two things:

- He forks the gem in his own repository and releases a new version.
- He packages the gem into the project. This effectively means that the gem becomes a plug-in and thus becomes manageable in the project itself.

It depends on the gem and type of change which option he chooses.

6.2.11 A new version of a gem, plug-in or the Ruby on Rails framework is released

While we feel that when a new version of a gem, plug-in or the Ruby on Rails framework is released, this should somehow be a trigger to let the customer know.

In general *Developer A* does not keep the gems or plug-ins up-to-date for a project. Also the version of the Ruby on Rails framework is usually not changed if a new version is released. Also when security fixes are released for the Ruby on Rails framework, no actions are taken. Only when the project actually suffers from the bug, upgrades are recommended to the client.

He thinks there are several reasons why there no action gets taken:

- Updating the Ruby on Rails framework gives a lot of problems with gems and plug-ins. So while a combination of certain gems and plug-ins works well on one version of Rails, it generates problems on newer versions. Thus updating Rails means also all gems and plug-ins have to be checked. The developer believes that if updating would be easier, projects would get updated more often.

  With Ruby on Rails version 3 updating gems should be somewhat easier, as all gems (often with compatible versions) are stored in a file in the project. Previously, in Ruby on Rails version 1 and 2 this was not the case.

- There is no good way of tracking security fixes in gems and plug-ins. The Ruby on Rails website generally gives information about security fixes, but this only includes the Ruby on Rails framework itself.

- There is no ticket created when a security fix for Rails comes out. Thus, the customer doesn't know about updates. *Developer A* also thinks that the customer may not care about updates. As long as the application works, the customer is not interested in newer versions of the

framework, as it generates no direct value for him.

We believe there is some room for improvement here. Creating a ticket takes relatively a short amount of time and gives the customer the choice of paying for the update or not. Also a different SLA model could be used in which updates are paid for.

6.3 Manager B

Manager B has worked approximately 2 years for the company. The work of *Manager B* is diverse; most of the time he's busy dealing with e-mails and various administrative tasks: time tracking, writing quotes and contracts, project management, and generating invoices (both regular projects and service level agreements); when there's time left, he develops code as well, which he finds very important as programming is the company's core business.

In the interview *Manager B* described the software development process roughly as displayed in Figure 10 below. Paragraph 6.3.1–6.3.9 (also mentioned in Figure 10) explain the development process in more detail. Paragraph 6.3.10–6.3.13 mention other aspects of the development process, not covered by the flowchart in Figure 10.

Besides the aspects of the development process, *Manager B* thinks other issues are important as well: e.g. finding new projects, finding new employees and keeping employees, but we regard these issues out of scope.

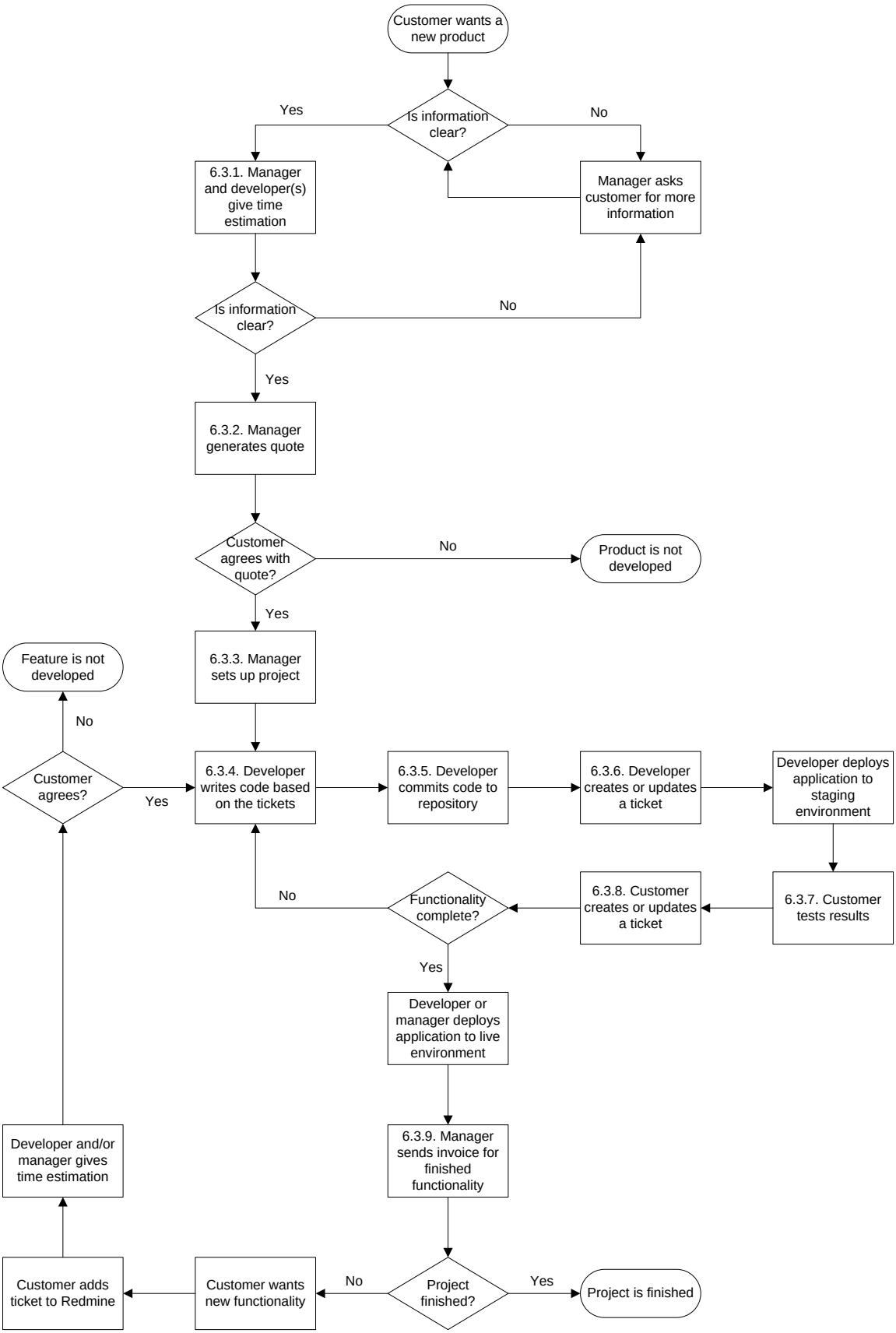

Figure 10: The company's software development process according to Manager B

6.3.1 Manager and developer(s) give time estimation

When a customer asks for a quote for a new project or new functionality, *Manager B* often asks a developer to estimate the development time for this functionality, besides himself. *Manager B* tries to spread enjoyable and less enjoyable projects over developers, which is difficult sometimes. He also tries to keep the same developer on the same project to ensure a high quality product and a satisfied customer, as the developer has more knowledge on how the project is organized and where what is located in the source code.

Manager B believes the following properties are expensive, both for the customer and the company, as the number of hours necessary to implement the functionality is always high and the risk of going over budgeted hours is also high:

- Everything that has to do with design (HTML and CSS) as well as technical implementations of everything that comes from designers. *Manager B* believes that if the company does the design during the development process, it is more efficient and cheaper for the customer.

- Integrations with other systems are expensive, especially SOAP integrations and instable APIs. Integrations in which information should be stored or synchronized are always more work to implement than information that should only be exposed or displayed.

Also, when the company is brought into the development process relatively late (e.g. *Project J*), the client does not always get the desired product for the price he wants. This happens often if the customer already has a designer that does not have web design as its core business, as the designer does not know the limitations of designing for the web, but also misses possibilities that are possible with web design. This lack of knowledge of designing for the web, as well as understanding that creating software is not a linear process (at least in agile settings) create situations where designers do not function well as project managers. Also when the company is brought in late, the deadlines are relatively short before the project should be finished.

Manager B thinks that the company should try to get into the development process earlier, as it is also capable of creating designs and doing consultancy. This would not only generate more revenue, but would also lead to a higher quality product for the client and a deadline that is more easily met.

Projects are taken over quite often. *Manager B* finds that if the company creates the project themselves from the start, it is less painful dealing with new features and the product is of higher quality. If a project is taken over, it is all about quality and responsibility, and also responsibility over the quality. Often there are bugs in the product that the customer did not expect. If the project has bugs at the start, it is hard to determine who is responsible. Responsibility is also hard to determine if the customer paid to invest in quality and tests, but there are still bugs in the system.

Five projects that were taken over did not have a very high code quality (*Project B*, *Project C*, *Project M*, *Project U*, and *Project W*). *Project P* was also taken over, but does have a test suite (test driven development was used by the original developers) and consequently the quality of *Project P* is considered adequate. Especially *Project B*, *Project C* and *Project M* are a real problem quality wise. This is mostly a problem for the customer, because he has a product of low quality and subsequent improvements to the code are expensive, as it takes more time to add improvements to low quality code. For projects that are taken over Manager B prefers time and materials based pricing over fixed price contracts.

Also, most of these projects do not have a test suite (so no test driven development was used). This may lead to regression when new functionality is added. Especially *Project M* suffers from regression due to parts of the system being connected, that should have no connection at all.

If the company is a subcontractor (*Project I*, *Project Y*, and *Project Z*), *Manager B* has the feeling that

the product is sold to the customer for a very high price. This also means that sometimes the company misses contracts as subcontractor, because the contractor tries to sell the product for a too high price to the customer, who in turn will choose a different implementation partner.

Also when the company works as a subcontractor, it should always have the right to say it made the product. This led to difficulties with the contractor in *Project Y* and *Project Z*.

For in-house projects (like *Project K* and *Project R*) *Manager B* recommends that the initial project should be kept simple and have more features added when necessary.

6.3.2 Manager generates quote

When *Manager B* writes a quote for a customer, he uses a recent quote for a similar project and copies usable parts to the new quote. Currently he does not have a template for writing a quote, although he would prefer this.

For the planning of the project, the company does have a template, which consists of 4 tables (in Excel):

- A table with features; for each feature one can add the estimated time and the sprint in which it will be implemented, showing the estimated time for each sprint.
- A table with configurations; the number of weeks per sprint can be configured, as well as the starting week and the number of weeks in the current year. The costs per hour can be configured, so based on the estimated time, the costs per issue are calculated; the costs per hour can be configured, so it calculates the costs per issue.
- A table with sprints (automatically generated); for each sprint a short description can be entered.

- A table with invoice moments.

Only the table with configurations is not copied into quote, while the other 3 tables are. If the customer accepts the quote, the features are copied to tickets in Redmine. *Manager B* would actually prefer to create parts of the quote in Redmine and then generate a quote based on these parts. This has the advantage that he wouldn't have to copy information from Excel to Redmine; in addition all the relevant information in the quote is actually already in Redmine, so for developers this would mean they have more easy access to all available information.

6.3.3 Manager sets up project

Previously the company had a separate mailing list for each project. *Manager B* doesn't think it's necessary to have a separate mailing list for smaller projects (e.g. if there is only one developer active on the project), although he doesn't mind either if other people want a separate mailing list. On the other hand, *Manager B* thinks it's difficult to manage a project if the customer and developer share information without letting the project manager know, especially considering additional functionality that the customer needs to pay for.

Manager B wants to have some sort of template or list for a project in Redmine, with tickets for all things that should be done when setting up a new project, among others: creating a repository, setting up a hosting account, asking for testimonials afterwards, creating a mailing list. This still allows managers or developers to skip certain steps if those aren't necessary, but it ensures no steps are forgotten. This would also allow new employees in the company to see what is necessary to start a new project.

Not only does *Manager B* want to standardize on the setup of projects, but on service level agreements (SLAs) as well. Some customers (*Project D*, *Project J*, *Project P*, *Project S*, and *Project U*) have a special status compared to other customers, especially when new bugs are encountered or features are requested. On the one hand these customers request more new features, so the

revenue is relatively large, but on the other hand more time is spend in communication, so the costs are also relatively high. Although in general larger projects form an excellent portfolio together, *Manager B* feels that the company should work towards one standard SLA for all customers in e.g. a fixed number of spendable hours, a staggered pricing in the number of hours, fixed pricing or time and materials based pricing, response time, duration of the contract, uptime for hosting; with a few configurable options. This would give the company better insight in what its rights and obligations are towards its customers.

6.3.4 Developer writes code based on the tickets

The wiki is mostly used for project specific information, like account and server information, and contact information. *Manager B* copied all information from DokuWiki to Redmine, so no information was lost. We are not sure everyone knows this information is there.

Manager B does not know exactly why Redmine was chosen over Trac, but he thinks it had to do with the fact that the company started doing its own hosting. Trac was not based on Ruby on Rails, while Redmine was, so it was possible for the company to host it themselves. *Manager B* also believes the interface and functionality of Redmine is better, than the interface and functionality of Trac.

If *Manager B* develops he usually uses Google and colleagues. He especially uses colleagues when things are hard to search for, e.g. if he does not know the name of certain functionality. He also uses RSS-feeds to keep up-to-date with the latest news on Ruby on Rails.

Some older products are hard to maintain (e.g. *Project D*), even though time is spent on refactoring. As new versions of Ruby on Rails are released frequently, projects age fast; this makes it difficult to maintain. People still call the company for new projects based on old products, but often it is just too old to use it as a basis.

6.3.5 Developer commits code to repository

Manager B thinks the switch from Subversion to Git was partly a hype, although looking back it wasn't a bad choice. Git feels better and more modern. It has conceptually an advantage, because it is decentralized, although the company doesn't really require this functionality. Git has a steeper learning curve than Subversion. On the other hand, Git has only one hidden folder in a project, while Subversion creates a hidden folder in each project. This makes moving code, while using Subversion, rather painful. Nowadays more files are automatically generated in projects, e.g. by using Sass (Catlin, Weizenbaum & Eppstein, 2010), which Git can work more easily with than Subversion.

6.3.6 Developer creates or updates a ticket

Manager B finds it a good thing that customers can see that developers create tickets, for exceptions for example. Even though the customer may not understand the actual issue, it provides transparency and the customer sees a pro-active response, which is very positive.

In general *Manager B* is quite satisfied with Redmine, but finds that grouping and sorting tickets doesn't always work as expected. Also, these grouping and sorting queries, e.g. a query for SLA periods, cannot be made available for all users, which makes it difficult to share views between employees. If *Manager B* discovers functionality that doesn't really work the way he would expect it to, he doesn't directly file a bug report with Redmine. On the other hand Redmine is actively developed, which is positive.

As Redmine is written in Ruby and as it is open source, the company can always adapt or write something they would like themselves. This also guarantees that the company can use Redmine in the future, so there is no risk of vendor lock-in.

6.3.7 Customer tests results

Customers differ a lot. Some customers are really active on project management themselves, while others need a lot of guidance by the company. *Manager B* thinks that customers who do not have experience with IT projects (*Project G*, *Project H*, and *Project T*) require the company to take a more

active role, although there are also customers with a high level of experience with IT projects that require a more active role by the company, especially if there are many things unclear during the project (*Project L* and *Project N*). If the company is not active enough projects may run over time or over budget (*Project L* and *Project N*).

If the customer is not involved enough in project management and when the company does not fulfill this role, one of the most important problems for the company is that the customer starts using and testing the product quite late (even though the product of functionality is finished for a few weeks already according to the company). So the customer (*Project S* and *Project T*) tests the product at a very late stage and then still expects the company to fix all the issues, while the company only gives a standard warranty of two months.

Manager B feels that if the customer has a lot of experience with IT projects and takes an active role, fixed price contracts are OK; otherwise he prefers time and materials based contracts. He also thinks that the company should determine how experienced the customer is at the start of the project, to avoid the above mentioned problems. The company could also do more consultancy work and project management for customers who do not have the ability or time to do it themselves, but fixed price would be hard in this case. All in all *Manager B* thinks the company should determine what kind of customer they're dealing with and tailor the project more to the client.

6.3.8 Customer creates or updates a ticket

Manager B finds it very useful that customers put tickets in Redmine. It ensures customers send less e-mails and provides an archive of issues. Some customers do not use Redmine often, but *Manager B* believes that if they want something badly, they will have to use Redmine anyway. Not all customers provide adequate information in the tickets, so *Manager B* says the company could hire someone (first line of support) who can improve the quality of the tickets by contacting the customer and who can handle phone calls of customers; this would allow developers to focus on the actual problem,

instead of figuring out what the problem was in the first place.

6.3.9 Manager sends invoice for finished functionality

MoneyBird is used for invoicing customers. Every month invoices are sent for finished functionality and service level agreements. Invoices are created based on the tickets in Redmine. *Manager B* would like the ability to automatically create invoices in MoneyBird for SLAs based on closed tickets in Redmine at the end of a month.

If the customer wants functionality next to the functionality stated in the quote, it is only billed if the estimated time (in the quote) has actually been spent or not. If it has, a new invoice is created for billing the additional functionality. *Manager B* ensures beforehand that it is clear for the customer what functionality requires additional payment and what functionality is covered by the initial quote.

SlimTimer is used by the management to discover billability for the active projects, although this functionality is fairly limited. Management puts the estimated time in the description of a task in SlimTimer, so it can be compared to the actual time. However, this is a tedious job if it should also be done for SLAs, as the estimated time changes frequently over the duration of the month; thus, it is often not done at all.

Manager B wants to find out if employees can track their time directly in Redmine (with help from a plug-in), to avoid the need of using a separate application (i.e. SlimTimer). However, it shouldn't be more cumbersome than SlimTimer is, to keep employees satisfied; if available plug-ins are not easy enough to work with, maybe the company should write a plug-in. Sherlock made it clear that usability is very important if you want people to work with the system, so if time tracking in Redmine is cumbersome it is no viable solution. Sherlock wasn't too complex, but the forms for tracking time were. At the time SlimTimer was a good choice, but now that the company has more projects, the overview becomes more difficult. It is not really clear which customer pays relatively much and which customer pays relatively little.

6.3.10 An exception is received by e-mail

When an exception is received, it depends on the project and how often it appears whether *Manager B* takes action. If he is active on the project, most of the time he creates a ticket for it.

The default warranty period that the company uses for its code is two months and is stated in all the quotes that it sends. It depends if the warranty expired whether the bug is immediately fixed (for free) or if a time estimation is given, although it also depends on the relationship with the customer and whether there is continuity in the project. If there is a lot of goodwill between the company and the customer (e.g. *Project X*), than the company is more eager to fix bugs for free, even when they are discovered outside of the warranty period. If contact between the company and the customer is tough, the warranty period is followed more strictly (e.g. *Project L* and *Project N*). The company's willingness to fix bugs after the warranty period also depends on whether the customer has a support contract or not. The company encourages its customers to sign a support contract, so the applications are kept up-to-date. This avoids the customer having to replace the application after several years due to a lack of maintenance.

Manager B also thinks that creating tickets out of exceptions could be automated, so developers won't have to create tickets for exceptions anymore.

6.3.11 Manager needs to generate report on worked hours

Manager B uses SlimTimer to generate weekly reports on hours worked by employees, which is sent by e-mail to the financial administration. These weekly reports have the days of the week on one axis and the employees on the other; totals are provided for the number of hours worked per day, per week and per employee. The latter is important for the financial administration, although *Manager B* does not know how this information is processed further (i.e. what systems are used).

6.3.12 Developer needs functionality that may be present in a gem or plug-in

Manager B uses external gems and plug-ins sparingly. Mostly he uses them based on past

experiences and whether they are new or not. If the gem or plug-in contains few lines of code or if the code is of bad quality, it is better not to use the gem or plug-in. *Manager B* finds that gems or plug-ins are often parts of code copied from a project, making them too specific with too few options.

6.3.13 A new version of a gem, plug-in or the Ruby on Rails framework is released

Only if the company actively works on a project, it is interesting to update gems or Ruby on Rails, but mostly only if new features are useful for the project. Except for security fixes maybe, *Manager B* believes gems shouldn't be updated for features or bugs pro-actively. Bugs only have to be fixed if they appear.

The company doesn't have a protocol for security updates (both in gems or Ruby on Rails) for communication with the customer. *Manager B* thinks for security the company could be more pro-active, for example keeping track of security patches.

6.4 Manager C

Manager C has worked approximately 3.5 years for the company. The work of *Manager C* is diverse; most of the time he's busy dealing with e-mails, phone calls and various other administrative tasks; he's also responsible for the company's hosting and various projects (most notably *Project D*).

Most of the e-mails that *Manager C* receives are questions by customers, updates on tickets by Redmine, server related information, and web server exceptions. Except for the first type, most of the e-mails he receives, he immediately throws away.

Manager C has many things on his to do list for which he has no time. *Project D* and hosting are difficult to combine, time wise. He has the feeling that he continuously runs behind schedule, although nowadays the days that he only spends answering e-mails are rare.

In the interview *Manager C* described the software development process roughly as displayed in

Figure 11 below. Paragraph 6.4.1–6.4.10 (also mentioned in Figure 10) explain the development process in more detail. Paragraph 6.4.11–6.4.15 mention other aspects of the development process, not covered by the flowchart in Figure 11. We also held an e-mail conversation with *Manager C*, mostly regarding the cause of the exceptions that are received by e-mail. The results of the e-mail conversation are stated in paragraph 6.4.12 and are merged with results from the interview.

Besides the aspects of the development process, *Manager C* thinks other issues are important as well: e.g. finding and keeping employees, and competence management; it is hard to get people. For this research we regard these issues out of scope.

Manager C thinks specific knowledge is very hard to hold on to. You can never replace people from one day on the other; new people need to orient themselves on new projects. Still, with Ruby on Rails the introductory period of projects is quite low.

Keeping knowledge on hosting is a much larger problem. This also means documenting knowledge and making sure knowledge is transferred is more important for hosting. Knowledge of code, e.g. gems, becomes outdated relatively fast (a year or so), so trying to keep or transfer knowledge on programming aspects is very difficult and time consuming. Keeping and sharing knowledge on how to tackle management problems would also be useful.

Manager C thinks code conventions could be useful, as well as conventions on what you should test and what not, when do you document code and what is acceptable without documentation, what are good names for methods. The lack of these conventions is noticeable when new inexperienced developers are hired, although they do pick up things fairly quickly from projects. Ruby kind of documents itself, in practice.

In case many new people are hired that would require training, then making a document with code conventions could be useful. For now it would mean a lot of work and a lot of arguing, resulting in a

lot of overhead.

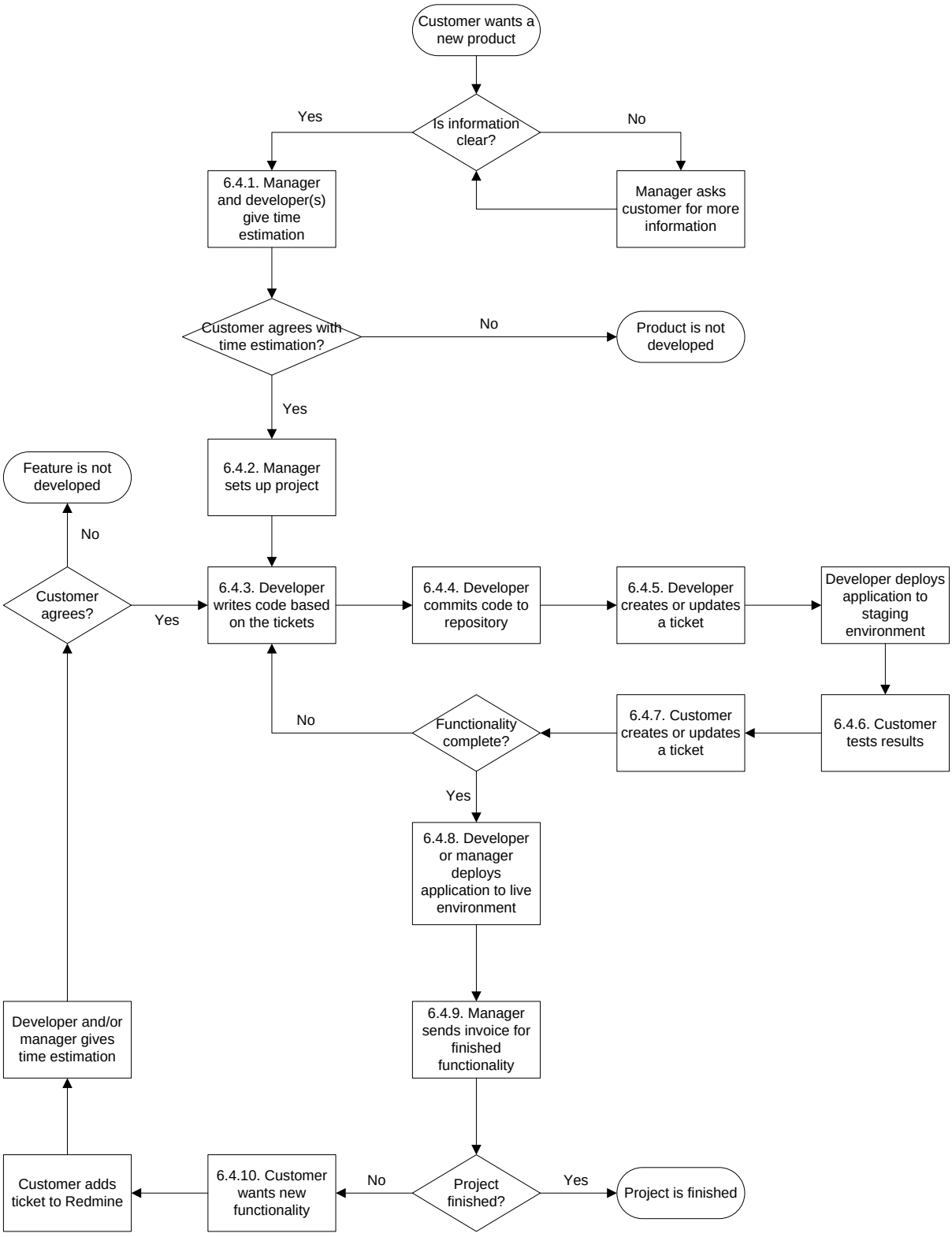

Figure 11: The company's software development process according to Manager C

6.4.1 Manager and developer(s) give time estimation

Manager C always tries to meet with new customers that call. Often he gets customers that "just want to know what it costs", while he feels he can't give an appropriate answer at that time. More details are necessary to be able to give an estimation of the development costs. If a customer doesn't want to meet, he tries to convince him to send information by e-mail, so he can give a better estimate.

When existing customers want new functionality or a new product, *Manager C* tries to get clear what they exactly mean, so he can estimate the required time in advance. If the customer puts information on the functionality in Redmine, *Manager C* can keep a better overview of all issues. Sometimes the customer communicates new functionality by e-mail, which means *Manager C* has to copy the information into Redmine. Whether the customer used Redmine or e-mail depends a bit on the customer; some customers feel that it is impossible to explain certain aspects in a ticket.

Depending on whether the customer agrees to the time estimation, the functionality is actually made.

The company can take over maintenance of existing Ruby on Rails applications very quickly without too much problems. Still, *Manager C* thinks you cannot take over a project blindly and say you will fix everything. Taking over a project should be seen as some kind of renovation, so it requires a few weeks work to get the quality up. If the customer doesn't want to invest this much time, then it is questionable whether the project should be taken over at all; especially because if you do take over the project and don't spend enough time in the beginning, the customer will never be satisfied with the quality of the project.

6.4.2 Manager sets up project

Manager C says that roles (especially project management) between customers and the company is not strictly defined or enforced, making the communication sometimes a bit unstructured. He thinks

that having a single contact on the customer's side and one on the company's side could improve communications. This doesn't mean all communication has to pass through one person; for example the ticket tracker still ensures communication can reach all people involved.

6.4.3 Developer writes code based on the tickets

If *Manager C* runs into a problem while programming, he uses Google or asks around in the office. *Manager C* thinks asking other people gives you new ideas and makes you pick a good solution. He never uses e-books and uses the wiki only for projects. On the wiki there are also some manuals for the office equipment, but the programming part is very outdated. *Manager C* doesn't really miss this information, as most employees are almost always in the office; also it requires a lot of time to keep the wiki articles regarding programming up-to-date.

In the past the company kept exactly how its server were setup (i.e. every command). This information is no longer usable, because for the next servers it is already outdated and slightly different tools are used. When dealing with hosting the commands you need to use are often the same, anyway.

Manager C likes it that the wiki is now incorporated into Redmine. Dokuwiki worked with configuration files, while in Redmine you can change settings in the application itself. This made working with Dokuwiki rather painful.

It is very useful that you can directly find information in the wiki if you already selected the project. On the other hand, if you selected a different project, it is quite cumbersome to see the wiki of a different project: first you have to browse to the project after which you can browse to the wiki. *Manager C* feels the latter situation is actually the most common one, so there is room for improvement there.

Considering project related information, the wiki is generally quite up-to-date. The information mostly lacking is information from customers on specific functionality; *Manager C* mostly realizes this

information is missing during programming.

Manager C only reads a blog or watches a part of webcast if he specifically needs to know something, and even then often only the code fragments; the webcasts mostly just take too long. There are no blogs that he follows actively.

Manager C seldom uses the source repository browser in Redmine; only for projects that he hasn't stored locally. He uses Git to checkout a different version of the code of the project, so he can examine older versions.

Manager C seldom reports bugs in Redmine or other systems. If he does report a bug, it is mostly for bugs in open source products that he has no clue about how to fix or circumvent; for example a segmentation fault somewhere. Sometimes the maintainers ask for more information, but he doesn't always know what they mean, and sometimes the maintainers don't do anything with the bug report. If the issue is not resolved, *Manager C* tries to find a different approach to the problem.

Manager C almost never reports bugs in gems or in Rails, although this depends a bit on the level of dependence on the needed functionality and whether *Manager C* is capable of fixing it himself. If he fixes it himself, he almost never reports the problem and solution, although he thinks it would be better if he did. The general problem is that the advantage of reporting a bug and solution is relatively low; the only advantage would be when later the gem needs to be updated in the project itself.

Manager C thinks this is kind of a shame, because not reporting a bug or solution doesn't make the product better. On the one hand people are complaining that some things don't work, but at the same time they are not helping with improving it either. On the other hand, it also often happens that if you report a bug or solution, no action is taken on the developer's side.

Manager C understands both viewpoints; as a developer of an open source project, often fixes are

committed of which the quality is really low, untested, undocumented without a clue what it is actually for.

If *Manager C* has a useful fix, he usually forks the project and puts it on GitHub. GitHub is a platform for hosting and managing public and private Git repositories, including collaborative development (GitHub, 2011).

6.4.4 Developer commits code to repository

Manager C has worked in the company with Git and Subversion. He prefers Git over Subversion, as it is more powerful. The largest advantage of Git is its flexibility; it has more possibilities to go back to older versions and to switch between versions. Git is focused on keeping multiple versions of projects. Especially if you are working on one project with more than three people, it is very useful to be able to create your own branches. Subversion is based on an older concept for that matter; keeping multiple branches of one project is not very easy.

Subversion is also used on the servers in hosting to keep the configurations up-to-date. This allows the Control Panel (accessible by customers) to change configuration files and ensures the servers always have the latest configuration files. In this case it is a completely linear process (i.e. there is only one branch), so there is no need to switch to Git.

The largest disadvantage of Git is the complexity; this is especially an issue when you just start using it. The Git commands are relatively long. In general *Manager C* is quite happy with Git. It is more complex, but it has more functionality, so this is kind of logical. The advantage of Subversion is that it breaks less easily, although canceling a commit halfway doesn't really work out well. In Git a lot of data can be recovered, but you really have to know what the commands are.

6.4.5 Developer creates or updates a ticket

Redmine is now setup primarily from a developer's point of view. *Manager C* thinks other fields, like how much it's going to cost, when it is finished and when it is invoiced could be added or used more

frequently.

The statuses *New* and *Agreed* are quite clear, but at the end of the development process it becomes a bit unclear. Currently *Resolved* and *Closed* are used. *Manager C* thinks *Closed* should mean it is done and paid for and one should probably never have to see the ticket again. Between *Resolved* and *Closed* there is room for improvement, for example *Online*, *Ready for testing*, *Approved*, *Live*, and *Invoiced*. The statuses should also be about going from tickets to invoices.

According to *Manager C* the move from Trac to Redmine was primarily due to the lack of possibilities to distinguish between customers and developers. Customers had always access to all tickets, which wasn't desirable at the time. Later in Redmine customers gain access to all tickets again. Trac was also hard to maintain, as it requires a separate instance of Trac for each project; secondly all configurations are located in configuration files, instead of being directly accessible from the interface. Functionality wise there is not much difference, but *Manager C* thinks Redmine is more flexible. Redmine was chosen over other potential ticket trackers, because it is written in Ruby.

Redmine is a little bit bloated. It has a lot of functionality; this is an advantage as well as a disadvantage. Sometimes you want to be able to do things, but it only works in certain situations, although in the last versions this is greatly improved. Especially moving or editing multiple tickets at the same time was a problem, but this is now possible.

For customers Redmine may be a little bit complicated. You see a lot of fields and it is not really possible to manage who sees what fields. That would be a nice improvement. Some parts of tickets are focused on customers; if a developer takes over, the content of the ticket switches to more technical aspects. The customer also gets an e-mail of these technical aspects; this may impede the use of Redmine by customers.

Manager C does take into account that customers can read all the tickets, language wise, although he does put technical details in the tickets. He thinks it's both an advantage and disadvantage that

you cannot keep tickets hidden from customers.

Manager C hasn't used any other ticket trackers besides Redmine in its projects.

6.4.6 Customer tests results

Manager C finds that customers don't always test everything. Sometimes new functionality is already online, but they still ask when it will be deployed. In other cases they do specifically mention the functionality is tested on the acceptance environment and can be deployed to the production environment. The latter works ideally.

What you do see is that some customers don't change the status of a ticket. For example, they respond to the ticket saying they agree with the time estimation, so a developer can create the functionality, but the customer doesn't actually change the status to *Agreed*. So adding new statuses may not help if they don't use them.

Another example is that customers can enter a date for a deadline, but they don't. *Manager C* thinks customers find all the fields just too complicated.

6.4.7 Customer creates or updates a ticket

If a customer enters a ticket, it depends whether *Manager C* immediately takes action. If the ticket is serious (i.e. high priority) or if he directly knows the answer, he immediately takes action. If the ticket concerns new functionality it takes more time and so he responds later. In general *Manager C* thinks it depends on the priority of the ticket how fast he responds.

Manager C sometimes reads tickets from other projects, although this depends a bit on the project. If the project is not really active or only in maintenance he is more inclined to read the ticket, than if the project is in active development. In the latter case he presumes someone else is working on the project and will act on the issue. It also depends on the number of tickets that are created or updated: if this number is small, *Manager C* reads the tickets more often, than if a large number of

tickets are created or updated in a short period of time.

Manager C thinks that open discussions can be done through e-mail, but for concrete tasks and issues he prefers to do business through Redmine; it gives a good overview of the project. Some things are still being communicated through e-mail, but *Manager C* has the feeling he sometimes loses information, although he does flag his e-mails sometimes. *Manager C* thinks that it is important that customers use only one communication channel. This ensures that you know where to look for information, instead of having to search in multiple channels.

The disadvantage of e-mail is that often only one person receives the information; this is especially a problem if that person is away for a longer time. An example was that a customer (*Project O*) sent an e-mail to *Manager C*, while he was on vacation, instead of putting the information in a ticket in Redmine. This meant the customer got a very late response. A slightly different example is that three employees of the same customer (*Project O*) each e-mailed the same question separately, without knowing that someone else already asked the question. In the beginning of the project, the customer (and also the customer of *Project D*) was skeptical about using Redmine, but now that they have seen it has advantages, they do use it.

According to *Manager C* customers use the medium of the “least resistance”, i.e. they prefer the medium on which they get the fastest response.

Manager C thinks that in default SLAs one could enforce the use of the ticket tracker. This doesn’t mean people can’t call or send e-mails, but it ensures that the company can say they only act on issues if they are reported in the ticket tracker.

6.4.8 Developer or manager deploys application to live environment

According to *Manager C* the largest problems in hosting are reliability and scalability. The current servers are becoming quite full and therefore slow. Lately the stability hasn’t been that good either. This is not really the quality the company wants to stand for and it should thus be solved. Solving it is

not really difficult, but past choices relating to the setup of the hosting environment resulted in more problems than it prevented. Afterwards it appears it could have been done better. The main problem is unreliable open source software, especially tooling that doesn't work as expected. The hosting platform is therefore a bit unreliable. It works, but the reliability becomes a bit critical.

Another problem is that the hosting platform is fixed to one Ruby version with a fixed set of gems. According to *Manager C* hosting independent of the Ruby version is the way to go. This means customers can have their own Ruby version and set of gems. This does mean the servers require more memory, but from a deployment viewpoint it becomes way more manageable and transparent. Now rarely an application breaks if a new version of a gem is installed on the server. This is not desirable. Another advantage of this setup is that if one application breaks for some reason, it would not influence any other application. Setting up this new environment, however, requires significant changes to the whole hosting environment.

6.4.9 Manager sends invoice for finished functionality

Manager C finds that calculating the overhead on each project takes too much time and therefore is often skipped. *Manager C* feels that he should be focused more on management, instead of writing code.

6.4.10 Customer wants new functionality

Manager C thinks SLAs can be improved by seeing maintenance of Ruby on Rails applications as a separate product, next to, among others, software development and hosting; not just as aftercare. The customer buys a maintenance service, which should be like a fixed amount per month for which the customer gets bug fixes, security fixes and Ruby on Rails updates, etc. For example, *Project J* already has this structure for a few hours per month.

6.4.11 An e-mail is received

Manager C thinks project specific mailing lists mainly result in overhead. Every employee was on

every mailing list. Currently there are only mailing lists for management, hosting, office, exceptions and projects. It gives overhead to create a list for every project, adding all employees, while in practice the list often falls into disuse. If you need to communicate to only a few people, you send them a private e-mail. If you have a more general question, you can send it to the office mailing list. *Manager C* thinks it only becomes an issue when the number of employees grows. Also part-timers know what's going on in the office if everyone gets the e-mails.

6.4.12 An exception is received by e-mail

Manager C receives so many e-mails that he throws most of the exception related e-mails away, without reading them. He often only responds if he just deployed a new version of a project or if he immediately knows how to fix it. If he has seen the exception before (i.e. it sparely appeared already earlier), which is most cases, he does not take any direct action. He also usually disregards exceptions generated by web crawlers.

Except for projects that are taken over without a test suite (like *Project M*), *Manager C* thinks most exceptions are generated due to a lack of proper tests. First of all this is caused by developers that have the feeling that writing tests takes more time than just writing code and secondly because developers only write tests for when the input is correct. He also has the feeling that in certain projects that are relatively old (e.g. *Project D*), the tests become somewhat outdated; thus developers do not run the test suite of this project.

Manager C external dependencies may also cause exceptions, but are relatively unimportant. He thinks sometimes the difference in development and production environment does result in problems.

Manager C thinks that having some kind of coding or testing guide, updating old test suites, and using continuous integration again, may solve most of the problems regarding the number of exceptions.

6.4.13 Manager needs to generate report on worked hours

Manager C finds SlimTimer to be way better than Excel sheets, but it is not the best solution possible. There are better tools, but these are missing the reporting functionality. Entering hours works OK, especially the timer works well. Entering hours afterwards (if the timer is not used) is relatively easy, but it could be a little better.

Generating reports on worked hours is OK as well; it does what we need. A downside is that the billability of projects is not really easy to see, but *Manager C* thinks this can really only be done if time is kept in a ticket tracker and if you can track your time really easy in it as well; especially if you can directly keep time at a ticket. For example: you open a ticket, start the timer, and when you're done, you can directly see how much time you spent on that ticket and how much time was estimated.

SlimTimer has the option to generate a report at the end of the month on how much time was spent on a project; if you sum the number of estimated hours in Redmine, you can compare these numbers, but it is far from ideal. If time tracking would be integrated into Redmine, there would be continuous insight into how a project is doing. Now it is only possible to see after another month has passed that, for example 140 hours were spent instead of 120. But why this was the case and on which tickets it was spent is very hard to deduct. It is impossible to find out on what issues more time was spent than estimated. In SlimTimer one can add tags, but *Manager C* finds that this doesn't really work either, as you don't know how much time was estimated for a certain tag.

Integrating time tracking in Redmine would require discipline, according to *Manager C*, especially starting and stopping the timer. Potentially one should be able to start a timer on the project itself and on some default tasks. Dropdowns as implemented in Sherlock do not really work well; *Manager C* says it must be single button to make it easy and part of an employee's routine. It is easy for everyone if one's hours are filled in automatically. And if you try to enter your hours later, it is often

impossible to tell on what issue exactly you sent how much time; in which case the hours should be entered on the project itself.

Manager C thinks it would be great if the company would gain insight into billability on a day-to-day basis. There is currently not enough insight in the status (costs and hours) of projects. Sometimes more time is spent on a project than invoiced, but it is not really known how much and on which projects.

According to *Manager C*, the reason Sherlock is no longer used is the interface that just wasn't user friendly enough for the most important tasks, namely adding hours. The project was actually just a bunch of CRUD-interfaces, but not really suitable for daily use. It has to better than this if you expect your employees to use it every day.

6.4.14 Developer needs functionality that may be present in a gem or plug-in

Manager C often uses external gems and plug-ins, although the results are variable. He does think that if someone else spent time on a certain solution, you can at least check out what the solution looks like. If it isn't satisfying, it is always possible to make it yourself. However, creating your own solution to a larger problem is always tougher than you think. The creator of the gem or plug-in often dealt with caveats that you still need to deal with, if you decide to write a solution yourself.

Manager C chooses to a use a gem partly on past experiences. Besides that he also looks at the commit history, the current activity of the project, how many people forked the project and what commits are in the forks. If the project itself is not really active, but a fork is, then the code if often still usable. It provides a good start, although you still need to write code yourself. If the gem or plug-in saves time, *Manager C* considers it useful.

In projects there are regularly problems with gems or plug-ins, especially if they deal with so called ActiveRecord models and mess with so called validations and filters.

If a gem or plug-in doesn't work as expected, *Manager C* first tries to make a workaround in the project itself, so the code of the gem or plug-in doesn't have to be changed. Sometimes however, the code is really bad and then you do have to change the code in the gem or plug-in itself. *Manager C* finds this kind of a shame, because it makes the project already less maintainable.

6.4.15 A new version of a gem, plug-in or the Ruby on Rails framework is released

Security fixes in Ruby on Rails and gems are rarely applied. *Manager C* thinks the largest problem is that a lot of stuff breaks on an upgrade. Gems are updated sparely. In practice minor updates of gems are automatically used, because the version of the gem is not so tightly fixed to the project. If there are really serious security issues, *Manager C* thinks updating is important, but otherwise not really.

Manager C doesn't create a new ticket if a new version of Ruby on Rails is released. The only way when upgrades are performed is if the customer pays and *Manager C* finds that the customer often doesn't see any value in upgrading. Customers mostly only value new functionality for the end user; technical improvements are not considered so valuable. Technical improvements are only interesting for the customer if it saves costs in the future.

Manager C thinks that updates of Ruby on Rails can be part of the SLAs; this means for a fixed price updates to new versions are always performed. Currently old versions often become unmaintainable. This is not a risk on the short term, but on the long term updating becomes a rather painful process. Especially for new developers older applications become a problem, because they learned to program in a newer version of Ruby on Rails.

Manager C thinks the only way to force an upgrade of Ruby on Rails is through an SLA. Some customers do see the necessity or advantages of upgrading (e.g. *Project B* and *Project X*), but most customers only look at the buttons on their screen. Those customers are often the same as the customers using outdated browsers. *Manager C* thinks this has to do with the mentality of the

customers and a lack of understanding computers.

6.5 Results

Based on results from the interviews we created a combined flowchart diagram in Figure 12. We also include a Data Flow Model (DFD) in Figure 13 to show the interaction each stakeholder has with the information systems in the company. We believe this gives some more insights next to the flowchart diagram.

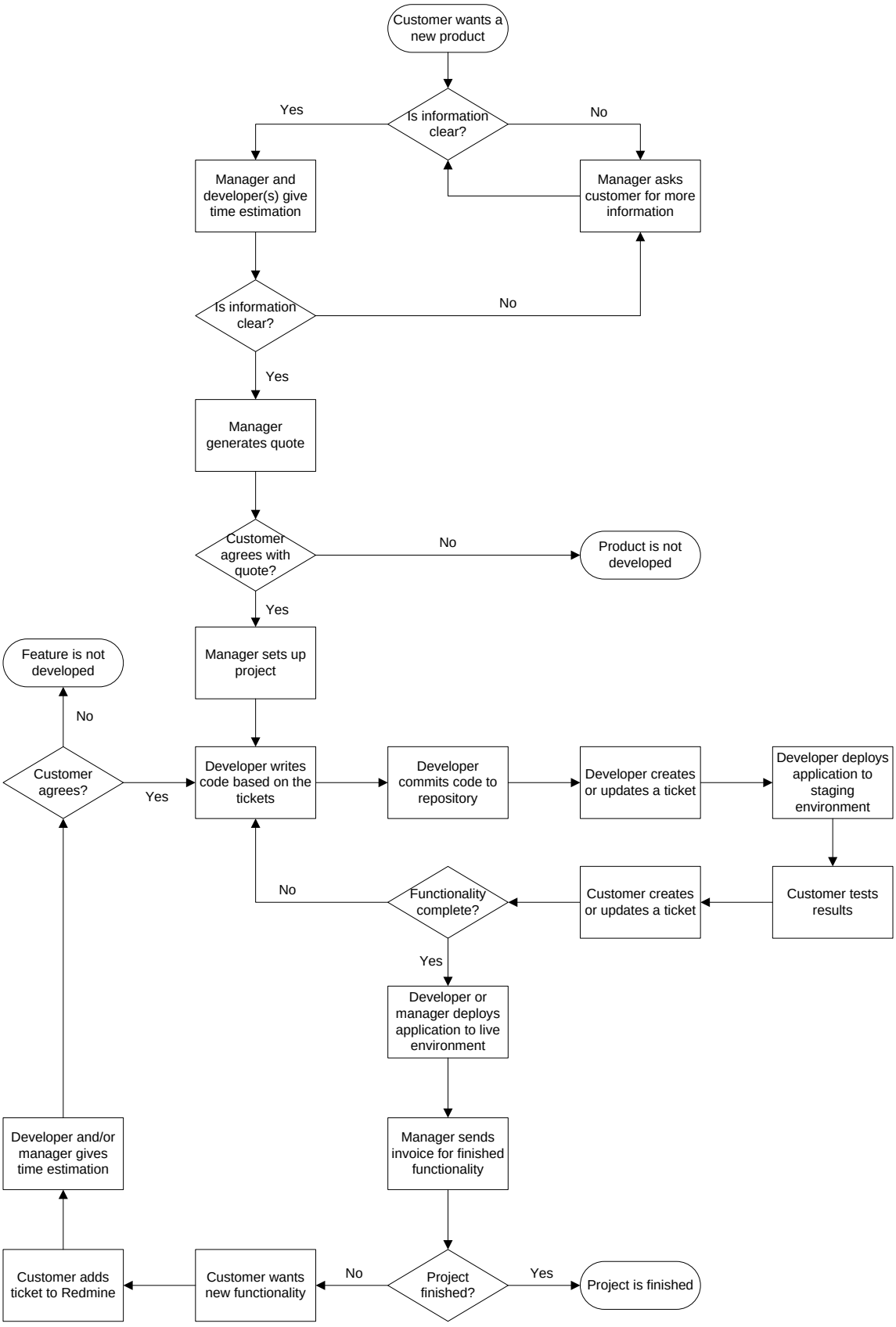

Figure 12: The company's software development process

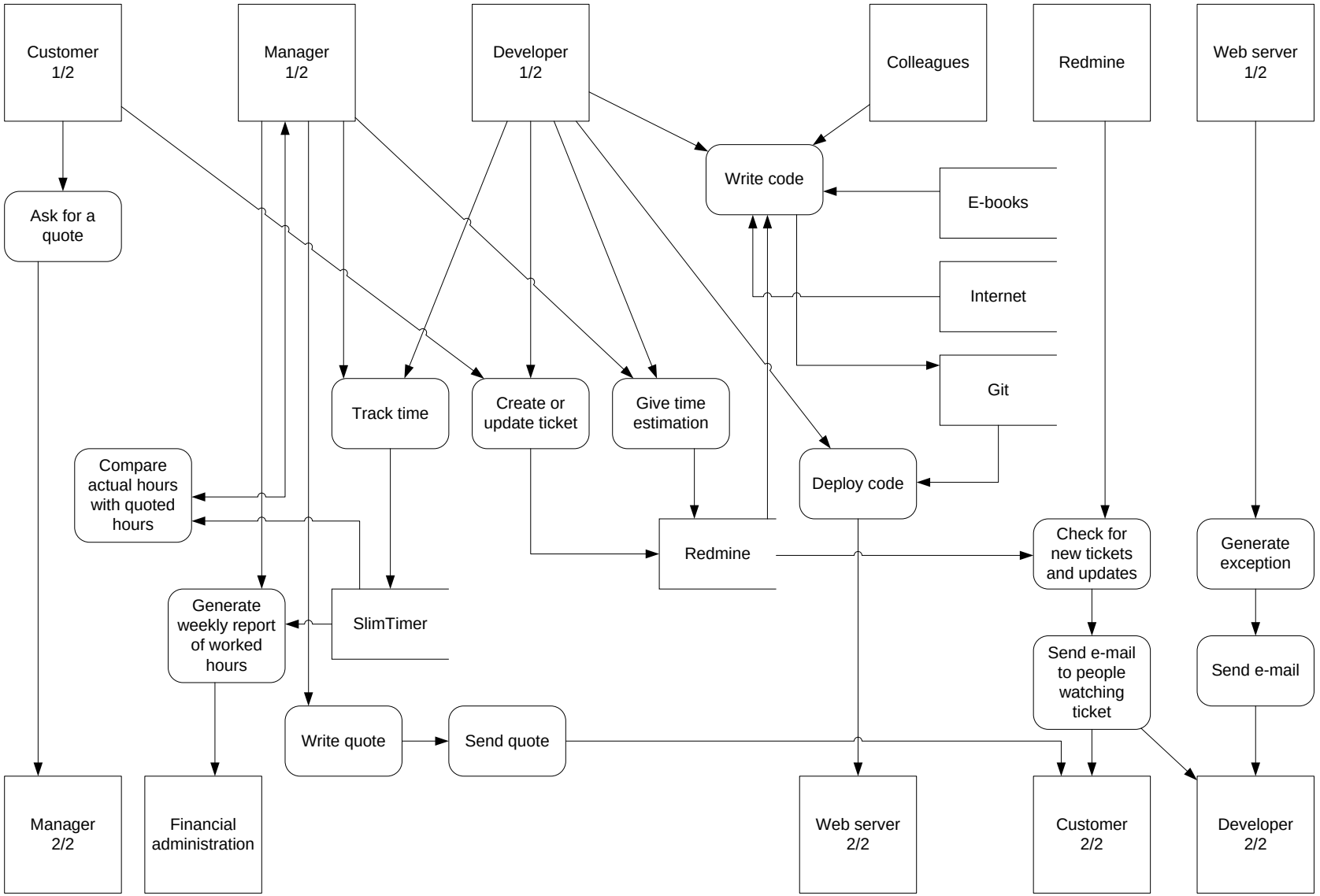

Figure 13: Data Flow Model of the company's development process

7 Data analysis

Based on the context of the company's projects (chapter 4), the documents it uses (chapter 5), and the interviews with relevant people (chapter 6), we analyze the problems the company is experiencing (7.1), analyze which patterns are used (7.2), which patterns can be used to counter the problems (7.3), and provide a long term solution for the company to improve its software development process by using patterns in its projects (7.4). The latter may also be interesting for companies in a similar context.

7.1 Problem bundle

To provide some structure to the problems discovered in the data and the interviews, we give an overview of the company's problems, below in Figure 14 in the form of a problem bundle. Although some problems regarding hosting are identified in the interviews, we leave most of these issues out of the figure, as we regard them out of scope.

The problem bundle is a descriptive overview of a company's problems. Daneva and Wieringa (2006) use a somewhat similar approach, called a problem dependency map. The problem bundle is also similar to a causal factor chart used in Root Cause Analysis (RCA). According to Rooney and Vanden Heuvel (2004), RCA is a four-step process involving the following: 1. Data collection; 2. Causal factor charting; 3. Root cause identification; and 4. Recommendation generation and implementation.

For this case study, we provide the collected data in chapter 4, 5 and 6. Instead of causal factor charting, we use the problem bundle shown in Figure 14; this figure also shows the root causes (depicted in green). Recommendations to the problems are given in paragraph 7.3; actual implementation of the recommendations is considered out of scope for this research.

While we do think that a causal factor chart, together with thorough root cause identification could have been used in this research, we feel that due the large number of problems, the problem

bundle's simplicity provides a better overview for the company's management to identify its problems.

Regarding syntax, all problems in the problem bundle are identified by rectangles; lines depict causalities. These causalities mean that if an underlying problem is solved or removed, the occurrence would be prevented, or reduced in its severity. Note that the empirical evidence for these causalities is largely based on data found in certain projects in the company. Thus these causalities cannot be considered scientific regarding other projects or companies. Also, some of the problems in the problem bundle are mere opinions by employees and thus should not be regarded as something more than that. And some proposed causalities might be more complex, than we discovered in this research. We do, however, think that a problem bundle provides some structure to the large number of identified problems. The colors of the rectangles depict the type of problem:

- Red rectangles show the main effects (or results) the other problems are causing. These are the issues that the company is suffering from the most.
- Yellow rectangles show problems largely due to external factors. These problems are very hard or largely impossible to change by the company itself.
- Green rectangles show the root problems. These are the problems we think the company's management should focus on. We recommend that the root problems are prioritized, so that the most important problems are solved first.
- Blue rectangles show all other related problems.

The numbers before each problem are used to trace the relations between the problems back to the data we collected; they have no meaning. A traceability matrix can be found in Appendix B: Problem bundle traceability matrix.

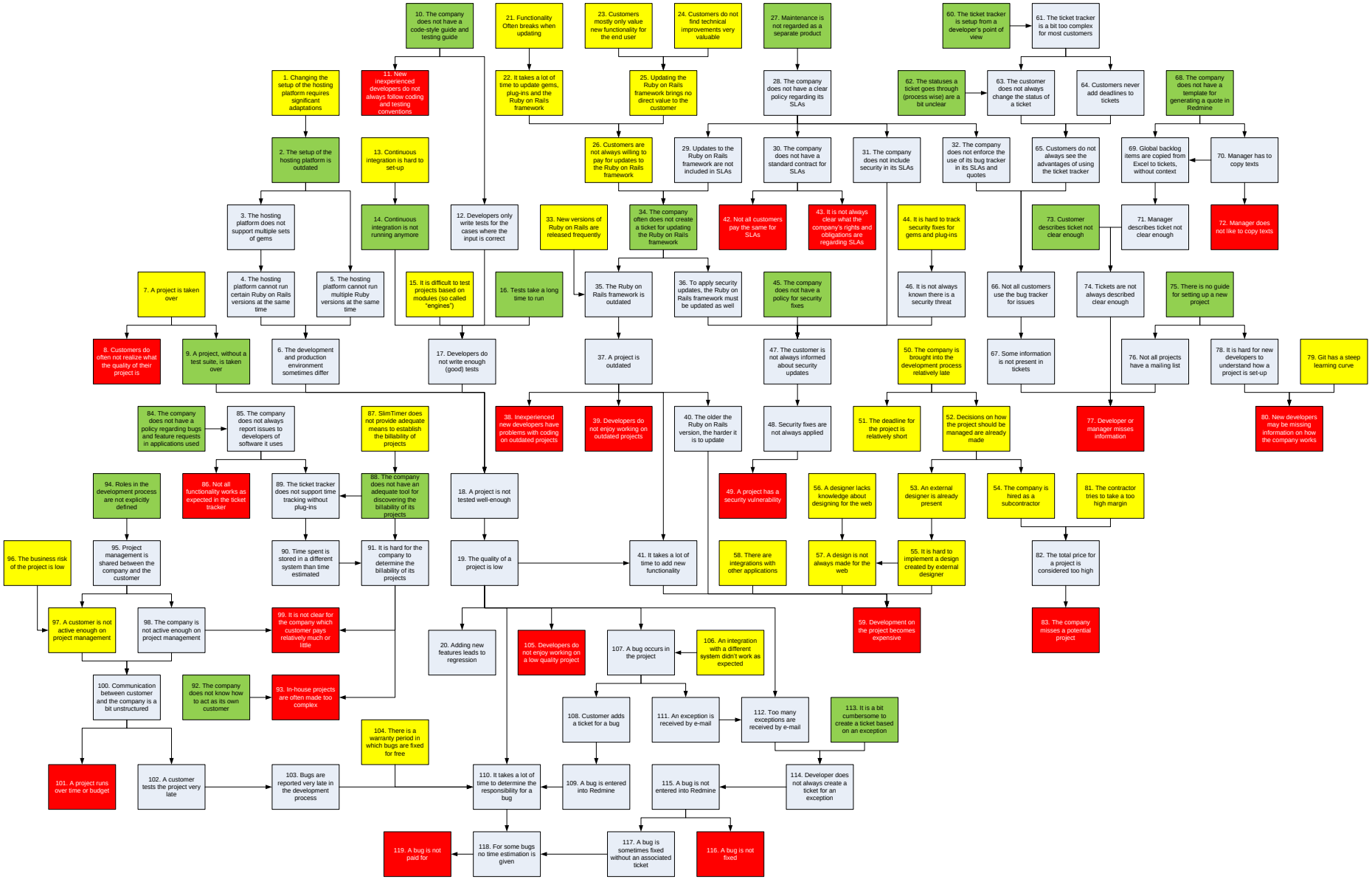

Figure 14: Problem bundle of the software development process in the company

7.2 Patterns overview

Below we give a list of patterns used in agile software development. This list is based on several sources: Coplien and Harrison (2004), Elssamadisy (2009), Beck (1999), Beck and Andres (2004), Shore and Warden (2007). The patterns are presented in random order.

The authors all provide some kind of structure to how patterns are related. We did not take this relation into account due to the amount of patterns involved. As the methods for determining the validity of patterns is rather scarce (see paragraph 3.6), we would also need to determine a method for establishing the validity of the relations between patterns.

While the patterns in these sources may not be exhaustive, we think it gives a good general impression of commonly used practices in agile software development.

The list gives the name of a pattern (including alternative names), the sources in which we found the pattern and whether we could establish if the pattern was used. The latter is done based on the context of the company's projects (chapter 4), its documents (chapter 5), the interviews with relevant people (chapter 6), and our own observations.

For this thesis we only list the patterns that we could measure or observe; based on the data we gathered we don't want to make assumptions on patterns for which we do not have enough data. Those patterns are therefore not included below. We also excluded the patterns from the Pragmatic Programming book as those patterns are on a lower level than we could discover in the data we gathered, although we do think that many of the patterns used in Pragmatic Programming are actually in use in the company. We believe research on a lower level could reveal the use of these patterns.

Pattern	Source	Used?
Size the schedule	Coplien & Harrison, 2004, p. 36 – 37	• No: developers are not rewarded for negotiating a schedule they prove they can meet. • Yes: external schedules are often two weeks longer than internal schedules.
Get on with it; partial evaluation	Coplien & Harrison, 2004, p. 38 – 41	• Yes: development is started as soon as some requirements are clear. In its quotes the company also writes that unclear requirements are made clearer during the development process.
Named stable bases	Coplien & Harrison, 2004, p. 42 – 43	• No: it is often not relevant, but in projects where it might be relevant, it isn't used.
Incremental integration	Coplien & Harrison, 2004, p. 44 – 45	• Yes: developers regularly commit code to the repositories, there are no restrictions; new versions of the applications are also often deployed, giving quick feedback. • No: during the development of *Project D*, developers often committed code only at the end of the day, leading to frustration due to failing tests if someone else committed just before.
Private world	Coplien & Harrison, 2004, p. 46 – 48	• Yes: developers have their own development environment. They can also create branches from the last version of an application and merge their changes later.
Build prototypes	Coplien & Harrison, 2004, p. 49 – 52	• No: prototypes are never made, although design mock-ups are sometimes made.
Completion headroom	Coplien & Harrison, 2004, p. 56 – 57	• Yes: headroom is calculated at the beginning of a project. • No: headroom is not calculated during the development process of a project.
Work split	Coplien & Harrison, 2004, p. 58 – 59	• Yes: work is often split into the smallest task for which a time estimation can be given, whether that work is urgent or not.
Recommitment meeting	Coplien & Harrison, 2004, p. 60 – 61	• Yes: we observed this in *Project D* and *Project R*. Besides that the office allows employees to talk to each other easily, so we noticed that meetings are often informal.
Work queue	Coplien & Harrison, 2004, p. 62 – 63	• Somewhat: work is divided into iterations and thus more critical tasks are done first; however work items in a single iteration are almost never given a distinct priority, except when it is a critical issue.
Informal labor plan	Coplien & Harrison, 2004, p. 64 – 65	• Yes: developers can make their own short-term prioritization plan, although new inexperienced developers are guided somewhat more.

Implied requirements; user story	Coplien & Harrison, 2004, p. 68 – 69; Elssamadisy, 2009, p. 149 – 152	• Yes: especially in quotes to customers implied requirement are often used. If the customer accepts the quote, the requirements are often made more explicit in the ticket tracker.
Developer controls process	Coplien & Harrison, 2004, p. 70 – 72	• Yes: the developer controls the process.
Work flows inward	Coplien & Harrison, 2004, p. 73 – 77	• Yes: customers create tickets, which developers can pick-up; often there is no interference from management.
Programming episodes; iteration	Coplien & Harrison, 2004, p. 78 – 79; Elssamadisy, 2009, p. 71 – 76	• Yes: work is almost always done in iterations (often two weeks). • No: when looking at a larger perspective (i.e. multiple sprints) we observed that in-house applications are made with functionality that isn't required at that time.
Someone always makes progress	Coplien & Harrison, 2004, p. 80 – 81	• Somewhat: tickets are always assigned to one person, so in that sense only one person deals with a single issue. On the other hand, this doesn't necessarily mean someone else is dealing with primary tasks.
Day care; progress team; training team	Coplien & Harrison, 2004, p. 88 – 91	• No: there is no single person in charge of all novices.
Mercenary analyst	Coplien & Harrison, 2004, p. 92 – 95	• No: if technical documentation is written at all, it is done by someone who is also involved in the development of that specific project (observed in *Project D*).
Apprenticeship	Coplien & Harrison, 2004, p. 108 – 109	• Yes: new employees have to follow a tutorial before they can start on real projects, although they may need more guidance after they completed the tutorial.
Solo virtuoso	Coplien & Harrison, 2004, p. 110 – 111	• Yes: observed in most projects (notable exceptions are *Project D*, *Project K* and *Project R*).
Engage customers	Coplien & Harrison, 2004, p. 112 – 115	• Yes: contact with customers is considered very important. Developers and customers can talk freely and there are often feedback moments. Customers can provide feedback whenever they want and developers are encouraged to help finding out what the customer really wants.
Surrogate customer	Coplien & Harrison, 2004, p. 116 – 117	• Yes: a surrogate customer was present in each in-house project (*Project K* and *Project R*).
Scenarios define problem; use case; stories	Coplien & Harrison, 2004, p. 118 – 119; Elssamadisy, 2009, p. 153 – 156; Beck & Andres, 2004; Shore & Warden, 2007	• No: use cases are rarely used. As far as we know only in *Project V* they were used.

Firewalls	Coplien & Harrison, 2004, p. 120 – 121	• Somewhat: managers deal with conflict situations with customers, if they arise. On the other hand, managers do develop themselves as well, making it hard for the managers to be tough on the customers.
Gatekeeper	Coplien & Harrison, 2004, p. 122 – 123	• No: we did not observe someone with the gatekeeper (or similar) role.
Self-selecting team; team continuity	Coplien & Harrison, 2004, p. 124 – 125; Beck & Andres, 2004	• No: teams are often based on the availability of people and the availability of people is often low.
Team pride	Coplien & Harrison, 2004, p. 128 – 129	• Yes: employees clearly feel they produce better software than other companies do, especially other technical third parties who write software they have to integrate with.
Patron role	Coplien & Harrison, 2004, p. 133 – 134	• Yes: the owner fulfills this role somewhat.
Matron role	Coplien & Harrison, 2004, p. 140 – 141	• Yes: someone with a matron role is present.
Wise fool	Coplien & Harrison, 2004, p. 148 – 149	• Somewhat: in general, developers do not mind speaking up to managers (or customers).
Moderate truck number	Coplien & Harrison, 2004, p. 155 – 157	• Somewhat: looking at software development, the company has a low truck number: not many people have critical domain expertise. When looking at hosting, the truck number is high: critical information regarding hosting resides with one person.
Failed project wake	Coplien & Harrison, 2004, p. 162 – 163	• No: *Project C* and *Project F* were ultimately canceled, but no 'wake' was held.
Developing in pairs; pair programming	Coplien & Harrison, 2004, p. 165 – 167; Elssamadisy, 2009, p. 223 – 227; Beck, 1999; Beck & Andres, 2004; Shore & Warden, 2007	• No: developing in pairs is rarely used.
Engage quality assurance	Coplien & Harrison, 2004, p. 168 – 170	• Yes: testing is part of the development process. Exceptions that users experience are automatically e-mailed to all developers.
Group validation	Coplien & Harrison, 2004, p. 174 – 175	• Yes: the initial (global) design is included in the quote; communication on architecture and design takes place through the ticket tracker.
Few roles	Coplien & Harrison, 2004, p. 180 – 181	• Yes: there are only a few roles in the software development process: developer / programmer and (project) manager; external roles are customer and more specifically (end) user.
Producers in the middle	Coplien & Harrison, 2004, p. 184 – 186	• Somewhat: developers get involved in all programming aspects of a project, but are not involved in all information on the project.
Stable roles	Coplien & Harrison, 2004, p. 187 – 188	• Yes: roles are stable for the duration of each project.

Conway's law	Coplien & Harrison, 2004, p. 192 – 193	• Yes: the organization is compatible with the product architecture. • No: the organization does not have periodic reviews of the product architecture or project management strategies.
Organization follows market	Coplien & Harrison, 2004, p. 197 – 198	• No: the company provides more types of services (consultancy, design, development, hosting, and maintenance), than clear responsibilities are defined within the development organization.
Face to face before working remotely	Coplien & Harrison, 2004, p. 199 – 201	• Yes: management always tries to meet with a customer, before the start of the project, except when the customer is located internationally.
Shaping circular realms	Coplien & Harrison, 2004, p. 204 – 205	• Yes: developers and managers are all located in the same office, encouraging communication. • Yes: Redmine allows all developers and managers to view all tickets of all projects.
Hallway chatter; co-located team; sit together	Coplien & Harrison, 2004, p. 213 – 216; Elssamadisy, 2009, p. 119 – 123; Beck & Andres, 2004; Shore & Warden, 2007	• Yes: developers and managers are all located in the same office, although customers are almost always located outside of the office.
The watercooler	Coplien & Harrison, 2004, p. 226 – 228	• Yes: there are many social structures in the company unrelated to workplace structures.
Coupling decreases latency	Coplien & Harrison, 2004, p. 231 – 233	• Yes: all central roles (developer, customer, manager) can communicate directly with each other. The use of a ticket tracker encourages sharing of information between all roles.
Architect controls product	Coplien & Harrison, 2004, p. 239 – 240	• No: the architect role is not specifically specified for a project.
Stand-up meeting; daily meeting	Coplien & Harrison, 2004, p. 247 – 249; Elssamadisy, 2009, p. 93 – 98	• No: rarely used nowadays. • Yes: known to have been used in *Project D* (initial development), *Project F* and *Project R*.
Architect also implements	Coplien & Harrison, 2004, p. 254 – 256	• Yes: although the architect role isn't explicitly defined, the person who architects the project also writes code.
Generics and specifics	Coplien & Harrison, 2004, p. 257 – 258	• Yes: experts design generic parts, if applicable.
Standards linking locations	Coplien & Harrison, 2004, p. 259 – 260	• Yes: the company prefers to use standards, especially for integrations, although the customer may sometimes decide differently.
Code ownership	Coplien & Harrison, 2004, p. 261 – 263	• No: in general everyone can change all code.
Feature assignment	Coplien & Harrison, 2004, p. 264 – 265	• Yes: the ticket tracker has the option to assign a ticket to a person. If the ticket describes a feature, that person is effectively responsible for implementing that feature.

Private versioning	Coplien & Harrison, 2004, p. 268 – 269	• Yes: developers can create local revisions, which they can later commit to the central repository.
Kickoff meeting	Elssamadisy, 2009, p. 77 – 80	• No: for most projects no official kickoff meeting has been held. • Yes: for *Project D* an official kickoff meeting was held. • Yes: a manager discusses a project with a developer before he writes a quote.
Backlog	Elssamadisy, 2009, p. 81 – 86	• Yes: a quote lists for each sprint which tasks are to be performed. • Yes: for *Project D* an official product backlog was kept by the ScrumMaster. • Yes: the ticket tracker lists the tasks for each milestone. • No: customers do not explicitly keep product backlogs.
Planning poker	Elssamadisy, 2009, p. 87 – 91	• Somewhat: planning poker is used irregularly in projects. In some projects planning poker is used for time estimations, while in others it is not.
Demo	Elssamadisy, 2009, p. 103 – 107	• Yes: in *Project D*, *Project F*, *Project K* and *Project R*. • No: in other projects.
Retrospective	Elssamadisy, 2009, p. 109 – 113	• No: retrospective meetings are not held anymore. • Yes: we found documents that retrospective meetings were held in the past.
Release often; daily deployment	Elssamadisy, 2009, p. 115 – 118; Beck & Andres, 2004	• Yes: new code for projects is deployed often to acceptance and live servers.
Self-organizing team	Elssamadisy, 2009, p. 125 – 129	• Somewhat: to some extend people can pick their own tickets, although it also happens that a manager decides who does what.
Cross-functional team; whole team	Elssamadisy, 2009, p. 131 – 135; Beck & Andres, 2004	• Yes: on most projects the team works together all the time; there are no hand-over moments.
Customer part of team; real customer involvement	Elssamadisy, 2009, p. 137 – 142; Beck & Andres, 2004; Shore & Warden, 2007	• No: for all external projects the customer isn't part of the team. • Yes: for in-house projects (*Project K* and *Project R*) the customer is part of the team.
Evocative document	Elssamadisy, 2009, p. 143 – 147	• No: if documentation is written at all (e.g. *Project D*) it is mostly representational, not evocative.
Information radiator; informative workspace	Elssamadisy, 2009, p. 157 – 160; Beck & Andres, 2004; Shore & Warden, 2007	• No: there is no information radiator.

Automated developer tests	Elssamadisy, 2009, p. 163 – 172	• Yes: on all new projects a test suite is available to all developers. Every developer can run the test suite to see what tests succeed and what tests fail. • No: on most projects that are taken over, no test suite is available.
Test-last development	Elssamadisy, 2009, p. 173 – 176	• No: instead test-driven development is used.
Test-first development; test-first programming; test-driven development	Elssamadisy, 2009, p. 177 – 182; Beck & Andres, 2004; Shore & Warden, 2007	• Yes: test-driven development is used on all new projects. On projects that are taken over no test-driven development is used.
Refactoring	Elssamadisy, 2009, p. 183 – 187; Beck, 1999; Shore & Warden, 2007	• Yes: refactoring is often used before new functionality is implemented.
Continuous integration	Elssamadisy, 2009, p. 189 – 196; Beck & Andres, 2004	• No: continuous integration is not used anymore. • Yes: continuous integration has been in use in the past, although due to maintenance problems, it no longer is.
Simple design; incremental design	Elssamadisy, 2009, p. 197 – 201; Beck, 1999; Beck & Andres, 2004; Shore & Warden, 2007	• Somewhat: in some situations the design is made more complex than necessary based on future expectations. In other situations the design is made as simple as possible, relying on future refactoring.
Functional tests	Elssamadisy, 2009, p. 203 – 217	• Somewhat: functional tests are often written, but they are mostly not written together with the customer.
Collective code ownership; shared code	Elssamadisy, 2009, p. 219 – 222; Beck, 1999; Beck & Andres, 2004; Shore & Warden, 2007	• Yes: on most projects all developers can change all code.
40-hour week; energized work	Beck, 1999; Beck & Andres, 2004; Shore & Warden, 2007	• Yes: working overtime is considered an exception; also developers and managers work in general no more than 8 hours per day.
Slack	Beck & Andres, 2004; Shore & Warden, 2007	• Somewhat: there is officially no time reserved for slack, but in general the schedule is not really tight, allowing developers to do other things, besides programming.
Ten-minute build	Beck & Andres, 2004; Shore & Warden, 2007	• No: at least *Project D* does not build in ten minutes. • Yes: some smaller projects
Shrinking teams	Beck & Andres, 2004	• Yes: at the end of the development phase, often some developers start working on a different project.
Code and tests	Beck & Andres, 2004	• Yes: only code and tests are maintained for a project.

Single code base	Beck & Andres, 2004	• Yes: in the cases of code duplication, projects are based on the same code base. • No: in the past various projects (*Project D*, *Project J*, *Project O*, and *Project S*) were all based on the same code, but had a different code base. This led to various problems with outdated projects.
Negotiated scope contract	Beck & Andres, 2004	• Somewhat: some contracts can be large from the start, so in that sense multiple contracts could be used instead. However, after the initial version of a project is released, the agreement between a customer and the company is often based on single tickets; thus effectively making a contract equal to a single ticket.

Table 3: Patterns, the source where the pattern is described, and its use in the company

7.3 Prioritizing and countering the problems

From the problem bundle (Figure 14), we deduct a list of root problems (the problems shown in green). These root problems can be prioritized according to various methods; we can ask stakeholders for their priority, for example by using the 100-Point Method (Leffingwell & Widrig, 2003); or we can count related problems in the problem bundle. Both have their advantages and disadvantages. Counting related problems is probably more objective than asking stakeholders. On the other hand, the problem bundle is a representation of the real world. Stakeholders may also hold information on how difficult or expensive it is to counter certain problems, while the problem bundle does not. Another disadvantage is that some related problems may be more detailed than others. Thus counting those problems places each of them on the same level. A combination of counting and asking stakeholders could be used as well.

For this research we choose to count related problems. We interviewed only three employees, so the results may give certain stakeholders an advantage if we would ask them for a priority; or alternatively if we would only ask those three employees, the results may be fairly biased. We believe counting related problems gives more reliable results in this case.

We can count the number of main effects (red rectangles) and related problems (blue rectangles)

that a root problem influences; fixing a root problem should result in a reduction in the associated main effects and related problems. We can also count the number of other problems that influence related problems of the root problem, which may diminish the effect of fixing a root problem. For this research we left the latter out of scope. More research is required in determining the effectiveness of fixing certain root problems in relation to other problems in the problem bundle. Maybe we could extend the relations in the problem bundle with estimated percentages of how much a certain problem influences another. This would allow us to calculate the effectiveness of fixing a certain problem on a related problem further down more easily and precise.

Below we give for each of the root problems one or more patterns that may be used to counter these problems. We also give a rationale to explain why the patterns may be a solution. The first two columns describe the number of main effects (**R**ed rectangles) and related problems (**B**lue rectangles) influenced by the root problem.

R	B	Problem	Pattern	Rationale
7	14	Maintenance is not regarded as a separate product	Organization follows market	Although this pattern may be useful for larger companies, we believe this pattern can still be useful, in the context of laying the responsibility for a specific service (consultancy, design, development, hosting, and maintenance) with one manager.
5	16	The company does not have a code-style guide and testing guide		
4	18	The setup of the hosting platform is outdated	Moderate truck number	We believe that having a shortage in people available for hosting results in the hosting platform becoming outdated, due to a lack of time. We recommend investigating a lower truck number for hosting; thus a higher number of people being capable of doing the same things.
4	15	Continuous integration is not running anymore	Continuous integration	Although it is difficult to setup and maintain, we still recommends using continuous integration again.
4	15	Tests take a lot time to run	Ten-minute build	We recommend making sure builds can be tested in ten minutes. If not, the tests (or code) should be refactored, so that they do run in ten minutes.

4	14	A project, without a test suite, is taken over		
4	7	The company often does not create a ticket for updating the Ruby on Rails framework	Code ownership Deploy along the grain; deploy people along the grain of the domain; one person/many hats	Nobody feels responsible for creating a ticket when a new version of Ruby on Rails is released; making someone responsible for the quality of a project, could potentially counter this effect. A different approach could be making an employee responsible for keeping the development framework (i.e. Ruby on Rails) up-to-date for a certain project, or possibly for more than one project. The latter means the employee has knowledge on problems that arise when updating the framework.
4	3	The company does not have a policy regarding bugs and feature requests in applications used	Code ownership	Nobody feels responsible for creating a ticket when a bug is encountered in an application used by the company (created and/or maintained by someone else). Making an employee responsible for a certain application may result in more bug reports being created, and thus potentially more bug fixes in applications used by the company. We do note that this is a kind of wide interpretation of the pattern.
3	7	Roles in the development process are not explicitly defined	Stable roles Move responsibilities	Because the roles in the software development process are not explicitly defined, the responsibility for certain issues shifts between someone from the customer and someone from the company. Using stable roles means also having the responsibilities straight, so the roles won't change overtime. Together with Stable roles, moving responsibilities can mean a decoupling between the customer's organization and the development organization, which in turn may improve the communication in the development process.
2	4	It is a bit cumbersome to create a ticket based on an exception		
2	4	The company does not have a template for generating a quote in Redmine		

2	3	The company does not have an adequate tool for discovering the billability of its projects	Planning poker	Although effective use of planning poker does not give insight in the billability of a project, we believe it does increase the billability.
2	2	There is no guide for setting up a new project	Kickoff meeting	Some kind of kickoff meeting with developers and managers would encourage them to think how the new project should be setup and how to divide responsibilities.
1	6	The ticket tracker is setup from a developer's point of view		
1	4	The statuses a ticket goes through (process wise) are a bit unclear		
1	2	The company does not have a policy for security fixes	Code ownership Deploy along the grain; deploy people along the grain of the domain; one person/many hats	Nobody feels responsible for creating a ticket when a security bug is encountered; making someone responsible for the quality of a project, could potentially counter this effect. A different approach could be making an employee responsible for security in a certain project.
1	1	Customer describes ticket not clear enough		
1	0	The company does not know how to act as its own customer	Simple design; incremental design Scenarios define problem; use case; stories	Applying the simple design practice would limit the scope and complexity of the project. If scenarios and use cases are written, they may limit the scope and complexity of the system.

Table 4: Root problems the company experiences and potential patterns that can counter them

Table 4 shows that we did not find a pattern for every problem. For some of those, the problem also describes the solution, e.g. "The company does not have a template for generating a quote in Redmine" would be solved by having a template for generating a quote in Redmine. For future uses of the problem bundle, we recommend writing problems in such a way that they do not include a solution in itself, allowing a less biased approach in finding patterns that may solve the problems.

7.4 Using patterns to improve the software development process

Besides using patterns to counter problems the company is experiencing, we also recommend that companies keep track of the patterns they currently use in the company, potentially for each project. By sharing the patterns among employees, (new) employees can learn more about the software development process of the company.

It also means employees can see for each project what patterns are used, and thus adjust their way of working, if required, and it means if employees run into a problem, they potentially can figure out how the problem was solved or prevented in other projects, thus enabling them to use existing practices in the company.

By making tacit knowledge explicit and letting existing employees list which patterns they use in which projects, the employees can see what the differences are in the use of practices between projects. By keeping track of the patterns, the employees can share knowledge on how the company works. We believe this kind of knowledge sharing can be done through a Wiki, although other software that enables knowledge sharing in the form of patterns can be used as well.

8 Conclusions and discussion

Based on the analysis of the data we gathered, this chapter presents our answers to the research questions stated in paragraph 2.3. Below we give conclusions to the research questions (8.1), we evaluate the validity of our research (8.2), we explain what the contribution of our research is (8.3), and finally we provide several directions for future research (8.4).

8.1 Conclusions

Below we answer each of our research questions. We include the answers to the subquestions together with the main questions.

1. *How are good practices used in the agile software development process? What do we mean with the agile development process in this research project? What do we mean with a good practice in this research project? How are good practices used in Scrum, Extreme Programming and Pragmatic Programming?*

In our research project we use the term 'agile software development process' to describe the collection of agile software methods and practices used by an organization. We use the term 'good practice' for practices that have empirical results from at least 10 companies and 50 projects, based on Jones (2000, 2009); we must note however that we find this definition rather limited validity wise. We also believe that a practice that includes information on the context in which it is applicable is more valid than a practice that does not. We deliberately avoid the term 'best practice' in our research, as it implies that there is no better alternative to the specific practice. Scrum uses various good practices, although it doesn't explicitly name them as such. Still, authors such as Beedle et al. (1999), and Shore and Warden (2007) name good practices used in Scrum. Extreme Programming includes values, principles and practices; Pragmatic Programming can be considered a list of good programming practices.

2. *How can we describe good practices in a uniform way?*

We used patterns to describe good practices in a uniform way. A pattern has a title, it includes information on the context in which it is applicable, it provides a description of the problem that arises in the context and a description of the forces that describe the problem, it gives a solution to the problem, and it explains why the pattern should be successful in countering the problem.

3. *How can we extract the use of good practices from project data? What methods exist for describing the context of projects? What methods exist for facilitating experience reuse among software managers and/or developers? How can we institutionalize the use of good practices in an organization and promote it among software managers and/or developers?*

We found that there are almost no standardized ways of describing the context of projects. Although it is specifically targeted at exploring interface problems of requirements engineering and architectural design, we found the method by Daneva et al. (2007) relevant for our research. The dimensions in this method provided an adequate overview of the company's projects and allowed us to compare the projects with each other. Still, we believe some further research is necessary to refine these dimensions, so we provided some hints for future research in paragraph 4.2.

Petter and Vaishnavi (2008) use narratives to facilitate experience reuse among software project managers, which they name Experience Exchange. We use a similar method as they do, but instead of using narratives we use patterns. Because these patterns are used by more practitioners in the software industry, it not only allows companies to share patterns internally, but externally as well.

We can institutionalize the use of these practices in an organization by letting the organization keep track of the practices it uses (and doesn't use) in each project. By encouraging managers and developers to share what selection of practices they use, other managers and developers can see how they can improve their use of practices, so that the overall software development process is

improved as well.

How can we improve the use of good practices in the agile software development process of a small sized company?

Based on the data we collected and analyzed, we recommend the following process for improving the use of good practices in the agile software development process:

1. Investigate which problems the company has and find out what the root causes of these problems are. We recommend using a problem bundle to visualize the dependencies among these problems.

2. Keep track of the good practices that the company uses and doesn't use. We recommend writing the practices in the form of patterns to keep a consistent structure between practices.

3. Try to match potential useful unused practices to the problems the company experiences and implement these patterns to solve the problems.

4. Have managers and developers share among each other which patterns they are using in which projects, through a knowledge management system, like for example a Wiki. This also allows new employees to quickly gain knowledge how the company works, and ensures that knowledge is kept in the organization if someone leaves.

8.2 Validity

We considered the possible validity threats (Yin, 2009) in this research. Below we give for each validity concern how and where we tried to address it to increase the quality of the research.

8.2.1 External validity

To improve external validity, Yin (2009) suggests using theory in single-case studies and using

replication logic in multiple-case studies. We have a single-case study, so we used theory. Our rationale for doing a single-case study can be found in paragraph 2.5.2 and 2.5.3.

If we look at the contributions of our research (paragraph 8.3), we feel that for each contribution external validity must be addressed. We therefore think that although we used theory to support our single-case study, we believe a multiple-case study may yield an improvement in external validity, as it means the contributions are tested in multiple companies.

However, we also note that according to research methodologists (e.g. Wieringa, 2010) we could generalize our conclusions based on a single case study. If we use Wieringa's mechanism-based generalization reasoning, then we could assume that our observations in the case study company could be similar to those that a researcher could make in other companies that share the same contextual settings as our case company. For example, we could think that our findings would also be observable in other small-sized agile companies in northern Europe that serve the same type of clients, work under the same type of contractual agreements, have similar work cultures, and face similar market challenges.

8.2.2 Construct validity

To improve construct validity, Yin (2009) suggests using multiple sources of evidence, establishing a chain of evidence, and having key informants review draft case study reports.

In our research we used literature and documents (and information systems), and held interviews with relevant people, so we did use multiple sources of evidence to support our research. We have also investigated multiple projects to establish the context the company is operating in. We do think some more interviews with relevant people would improve the quality of our constructs.

By using a traceability matrix (Appendix B: Problem bundle traceability matrix) we created a chain of evidence, thus allowing researchers to find what problem was discovered in which source(s) of evidence. We do think that mentioning more explicit what we searched for in what databases would

make the chapter on literature (chapter 3) more useable by others, so we feel that the construct validity of the literature part could have been higher.

Thirdly, we gave each interviewee the opportunity to comment on the results and we provided the managers the opportunity to comment on the overall report.

8.2.3 Internal validity

Internal validity is mostly dealt with during data analysis. To improve internal validity, Yin (2009, p. 41) suggests doing pattern matching, doing explanation building, addressing rival explanations, and using logic models. Yin (2009, p. 42–43) writes that internal validity is mainly a concern in explanatory case studies, and is of less importance in descriptive or exploratory studies. As our research is largely exploratory, we do not consider internal validity equally high important compared to other validity threats.

We consider patterns matching and logic models largely irrelevant for this research.

Yin (2009, p. 141) suggests using a parallel procedure to explanation building for exploratory case studies based on *The discovery of grounded theory: Strategies for qualitative research* by Glaser and Strauss (1967). The procedure is part of a hypothesis-generating process aimed at developing ideas for further studies. We have not used this procedure, but do recommend evaluating its use in future research.

We consider time-series analysis also largely irrelevant for this research, but note that in chapter 4 we did not include the period in which the projects were executed as a dimension. We believe including the timeframe in future use of these dimensions may lead to different results, for example due to the availability of new technology.

8.2.4 Reliability

To improve the reliability of the research, Yin (2009) suggests using a case study protocol and

developing a case study database.

We did not explicitly use a case study protocol; instead we used the protocol for a research as suggested by Verschuren and Doorewaard (1998), as we are more familiar with its layout. We do believe that using the case study protocol as suggested by Yin (2009) may have improved the reliability of our research.

We did not develop an explicit case study database, so we could have improved our research on that matter. However we did include all data regarding the context of the company's projects (Appendix A: Project dimensions); we also explicitly listed what documents we found (chapter 5) and how we collected these documents (paragraph 5.1); and we recorded all interviews allowing other researchers to listen to those interviews as well.

We think keeping track of all documents and having all interviews on paper would improve the reliability of our research.

8.3 Contributions

The contribution of our research is two-fold: on the one hand we give managers and practitioners deliverables that they can use, while on the other hand we provide a process that allows them to find potential solutions to problems they experience.

8.3.1 Deliverables

1. A problem bundle allows managers and practitioners to see relationships between problems and to identify potential root problems. It gives them a visual insight into which problems to tackle.

2. A table of good practices allows practitioners (in this case developers and managers) to gain insight in what practices are used in an organization and in which way. It also shows which practices are not (completely) used in the organization. This makes these unused practices at

the same time potential solutions to problems experienced in the organization.

8.3.2 Process

Besides the deliverables our research also gives practitioners a process, which they can use to identify problems in an organization, find deeper causes to these problems, identify practices used in the organization, and identify practices that can serve as potential solutions to these problems:

1. Use documents, project dimensions and interviews with stakeholders to identify problems in the organization.
2. Use a problem bundle to find deeper problems that may cause the problems that are experienced.
3. Identify practices used in the organization based on the documents, project dimensions and interviews.
4. Identify practices that can be used as potential solutions to counter the problems experienced.
5. Apply useful practices.
6. Evaluate the use of the practices.

After the evaluation, the practitioner can continue the process by identifying new problems. In Figure 15 below, we give a visualization of the process.

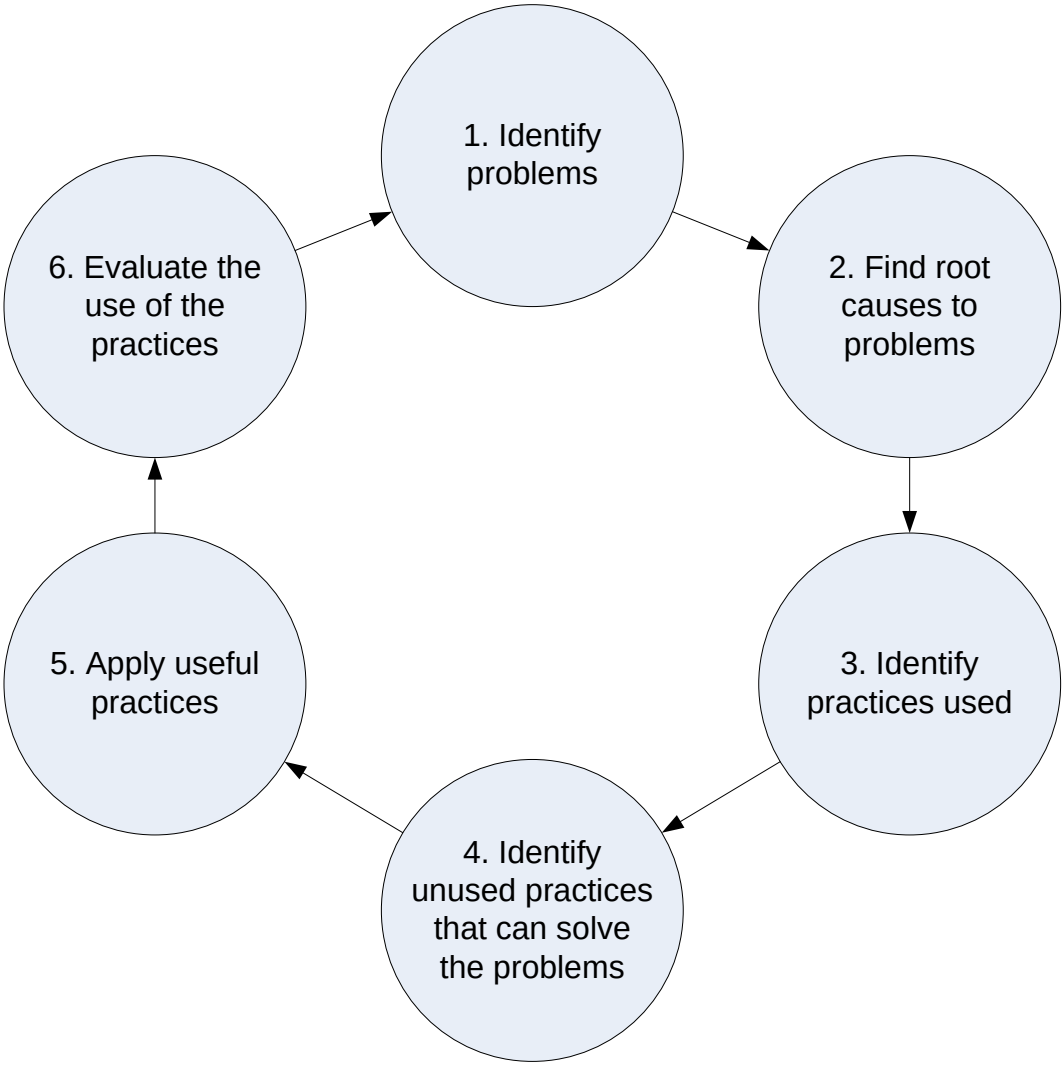

Figure 15: A process for countering identified problems with good practices

8.4 Future research

Based on our research we recommend the future research into the following aspects.

8.4.1 The use of good practices in other market sectors

In our research we only focused on the use of practices in (agile) software development (paragraph 3.4). It would be interesting to see how other market sectors deal with good practices, how good practices are determined, how good practices are shared among practitioners and how good practices are shared inside companies.

8.4.2 Determining the validity of good (or best) practices

In paragraph 3.6 we only found one method for determining the validity of good or best practices in software development, which we find fairly limited, both in number and in validity of the method itself. We recommend doing more research in improving determining the validity of good or best practices. By improving the way to determine the validity of good or best practices practitioners can rely more on these practices to improve their development process.

8.4.3 Validation of the problem bundle

We recommend further research on the validation of the use of problem bundles (paragraph 7.1), in the company we did the research, in other companies doing software development, and in companies in other market sectors. Validation of the use of problem bundles is important to make it a useful tool for practitioners.

Not only the validity of the use of the problem bundle is important, as well as the validity of the relationships between the problems. More research is required to improve the quality of a problem bundle for practitioners.

8.4.4 Improving the dimensions of a project's context

In chapter 4 we used 20 dimensions to describe the context of a project. In paragraph 4.2 we wrote several suggestions for future research on these dimensions. We believe the list of 20 dimensions may not be exhaustive and additional dimensions may be uncovered in future research. We do think that some standard way of describing a project's context would improve the ability to compare projects to each other, and thus additional research into this field is important.

Besides the dimensions it would be interesting to find out if we could use a combination of problems found and patterns used in a project to describe its context. This should allow companies with projects that have roughly the same context to select the same patterns to improve their chances of a successful project.

8.4.5 Determining the priority of problems in the problem bundle

The problem bundle in paragraph 7.1 uses colors to distinguish types of problems, but it does not provide practitioners with functionality to prioritize between problems. It would be interesting to include the priority of problems in the problem bundle and to allow practitioners to easily calculate or see what problems they should focus on first.

8.4.6 Sharing used patterns

In paragraph 7.4 we recommend the use of a system, such as a Wiki, to allow developers and managers to share and keep track of used good practices in the form of patterns. More research in this area is necessary to ensure developers and managers actually use the system and to ensure sharing and using patterns doesn't become cumbersome.

8.4.7 Internal validation of using practices to solve problems

We provided several suggestions in paragraph 7.2 for using patterns to solve certain root problems. Further research is necessary to see whether these suggestions are correct (and thus if using the patterns actually solve the problems). It would also be interesting to see if solving the root problems also solves other problems in the problem bundle.

References

- Alexander, C., Ishikawa, S., & Silverstein, M. (1977). *A Pattern Language: Towns, Buildings, Construction*. USA: Oxford University Press.
- Alexander, C. (1979). *The Timeless Way of Building*. USA: Oxford University Press.
- Alexander, I., & Robertson, S. (2004). Understanding Project Sociology by Modeling Stakeholders. *IEEE Software, 21(1)*, 23–27.
- Amrit, C. (2008). *Improving Coordination in Software Development through Social and Technical Network Analysis, PhD thesis series number 08-134*. Enschede: CTIT
- Andringa, S. (2008, June 17). *Een introductie tot agile softwareontwikkeling en een pragmatische evaluatie van de agile ontwikkelmethode Scrum*.
- aTech Media (2010, November 24). Git, Mercurial & Subversion Hosting with project management baked in. In *Codebase*. Retrieved November 24, 2010, from http://www.codebasehq.com.
- Basili, V. R., Caldiera, G., & Rombach, H. D. (1994). *The Goal Question Metric Approach*.
- Bates, R. (2010, December 9). Free Ruby on Rails Screencasts. In *Railscasts*. Retrieved December 9, 2010, from http://railscasts.com.
- Baudis, P., & Chacon, S. (2010, November 2). Git is…. In *Git - Fast Version Control System*. Retrieved November 24, 2010, from http://git-scm.com.
- Beck, K. (1996). *Smalltalk Best Practice Patterns Volume 1: Coding, draft*.
- Beck, K. (1999). *Extreme Programming Explained: Embrace Change*. Addison-Wesley.

- Beck, K., & Fowler, M. (2000). *Planning Extreme Programming*. Addison-Wesley.
- Beck, K., & Andres, C. (2004). *Extreme Programming Explained: Embrace Change, Second Edition*. Addison-Wesley.
- Beedle, M., Devos, M., Sharon, Y., Schwaber, K., & Sutherland, J. (1999). *SCRUM: An extension pattern language for hyperproductive software development*.
- BlueTools (2010). De snelste manier om facturen te maken. In *MoneyBird*. Retrieved January 13, 2011, from http://www.moneybird.nl.
- Catlin, H., Weizenbaum, N., Eppstein, C. (2010, December 17). Sass makes CSS fun again. In *Sass – Syntactically Awesome Stylesheets*. Retrieved February 3, 2011, from http://www.sass-lang.com.
- Coplien, J. O., & Harrison, N. B. (2004). *Organizational Patterns of Agile Software Development*. Castleton, NY: Prentice Hall.
- Daft, R. L. (2001). *Organization Theory and Design, Seventh Edition*. Cincinnati, OH: South-Western College.
- Daneva, M., & Ahituv, N. (2010). *What Practitioners Think of Inter-organizational ERP Requirements Engineering Practices: Results from a Focus Group*.
- Daneva, M., Hordijk, W., Racheva, Z., & Wieringa, R. (2007, June 15). *A Dimensional Analysis of Software Projects, draft 10.0*.
- Daneva, M., Wieringa, R. J. (2006, May 5). A requirements engineering framework for cross-organizational ERP systems. In *Requirements Engineering, 11*, 194–204. London: Springer-Verlag.
- DeGrace, P., & Stahl, L.H. (1990). *Wicked Problems, Righteous Solutions: a Catalogue of*

Modern Software Engineering Principles. Yourdon Press.

- Dybå, T., & Dingsøyr, T. (2008). Empirical studies of agile software development: A systematic review. *Information and Software Technology*, 50, 833–859.

- Edgewall Software. (2010, January 31). Welcome to the Trac Open Source Project. In *The Track Project*. Retrieved February 17, 2010, from http://trac.edgewall.org.

- Elssamadisy, A. (2009). *Agile Adoption Patterns: A Roadmap to Organizational Success*. Addison-Wesley.

- Github. (2011, April 19). Secure source code hosting and collaborative development. In *GitHub*. Retrieved April 19, 2011, from http://www.github.com.

- Gohr, A. (2010, August 29). DokuWiki. In *DokuWiki*. Retrieved November 24, 2010, from http://www.dokuwiki.org/dokuwiki.

- Hansson, D.H. (2009, October 18). Web development that doesn't hurt. In *Ruby on Rails*. Retrieved October 18, 2009, from http://rubyonrails.org.

- Hooimeijer, P., & Weimer, W. (2007, November 5-9). Modeling Bug Report Quality. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, 34–43.

- Hunt, A., & Thomas, D. (1999). *The Pragmatic Programmer: from journeyman to master*. Addison Wesley.

- Jones, C. (2000). *Software assessments, benchmarks, and best practices*. Harlow: Addison Wesley.

- Jones, C. (2009). *Software engineering best practices: lessons from successful projects in top*

companies. New York, NY: McGraw Hill.

- Kerievsky, J. (2000). *Patterns & XP*. Industrial Logic.

- Kitchenham, B. A., Pfleeger, S. L., Pickard, L. M., Jones, P. W., Hoaglin, D. C., El-Emam, K., & Rosenberg, J. (2001, January). *Preliminary Guidelines for Empirical Research in Software Engineering*, National Research Council of Canada.

- Lang, J.P. (2010, January 20). Redmine - Overview. In *Redmine*. Retrieved February 17, 2010, from http://www.redmine.org.

- Laudon, K. C., & Laudon, J. P. (2000). *Management Information Systems: Organization and Technology in the Networked Enterprise, Sixth Edition*. Upper Saddle River, NJ: Prentice-Hall.

- Leffingwell, D., & Widrig, D. (2003). *Managing Software Requirements: A Use Case Approach, Second Edition*. Boston, MA: Addison-Wesley.

- Martin, R. C. (2000, July/August). eXtreme Programming Development through Dialog. *IEEE Software, 17*, 12–13.

- Petter, S. (2008). A Process to Reuse Experiences via Written Narratives among Software Project Managers: A Design Science Research Proposal. In V. K. Vaishnavi, & W. Kuechler Jr. (Eds.), *Design Science Research Methods and Patterns: Innovating Information and Communication Technology*, 41–54. New York, NY: Auerbach.

- Petter, S. & Vaishnavi, V. K. (2008, April). Facilitating experience reuse among software project managers. In *Information Sciences: an International Journal, 178(7)*, 1783–1802. New York, NY: Elsevier.

- Rising, L., & Janoff, N. S. (2000, July/August). The Scrum Software Development Process for Small Teams. *IEEE Software, 17*, 26–32.

- Rooney J. J., Vanden Heuvel, L. N. (2004, July). Root Cause Analysis For Beginners. In *Quality Basics*, 45–53.
- Schwaber, K. (1996). *SCRUM Development Process.*
- Schwaber, K. (2004). *Agile Project Management with Scrum.* Redmond, WA: Microsoft Press.
- Shore, J., & Warden, S. (2007). *The Art of Agile Development*. Sebastopol, CA: O'Reilly.
- Statistics Canada (2010, June 17). North American Industry Classification System (NAICS) 2007. Retrieved December 8, 2010 from http://www.statcan.gc.ca/subjects-sujets/standard-norme/naics-scian/2007/list-liste-eng.htm.
- Sutherland, J. (2001). Agile Can Scale: Inventing and Reinventing SCRUM in Five Companies. *Cutter IT Journal, 14(12)*, 5–11.
- Takeuchi, H., & Nonaka, I. (1986). The new new product development game. *Harvard Business Review*, 137–146.
- The Apache Software Foundation (2010, November 24). Apache Subversion Features. In *Apache Subversion*. Retrieved November 24, 2010 from http://subversion.apache.org/features.html.
- Van Solingen, R., & Berghout, E. (1999). *The Goal/Question/Metric Method: a practical guide for quality improvement of software development*. London: McGraw-Hill.
- Verschuren, P., & Doorewaard, H. (1998). *Het ontwerpen van een onderzoek, tweede druk*. Utrecht: LEMMA.
- Wieringa, R. J. (2011). The Structure of Design Theories, and an Analysis of Their Use in Software Engineering Experiments. *International Symposium on Empirical Software*

Engineering (ESEM), accepted for publication. Banff, Canada.

- White, R. (2006, July 20). SlimTimer launched. In *SlimTimer Blog*. Retrieved November 24, 2010, from http://blog.slimtimer.com/2006/07/20/slimtimer-launched.

- Yin, R. K. (2009). *Case Study Research: Design and Methods, Fourth Edition*. Thousand Oaks, CA: SAGE.

Appendix A: Project dimensions

Below we define the dimensions for each project in the company, based on Daneva et al. (2007). To be able to refer to each project individually, we assigned each project a different letter, which is given above each table. In the bullet points below we state some recommendations for three of the dimensions, so the dimensions can be filled out more easily for the projects.

- For the dimension *Procurement process* (7), we use the nominal scale (None, Proprietary, Government, and Military) as proposed by Daneva et al. (2007), but added the 'Unknown' value, if we do not know if there was a procurement process at all. We also try to be as specific as possible, so we also list the information below if applicable. Although this information may not necessarily influence the requirements engineering process or architectural design of the project, we believe it gives an overview how the company gets its projects.
    - Tender: the vendor was selected based on a tender
    - Business network: the vendor was selected based on knowing someone in the business network (via-via)
    - Previous experience: the vendor was selected based on previous experience
    - Neighborhood: the vendor was selected because of its physical proximity
- For the dimension *Business sector* (9), we use the North American Industry Classification System (NAICS) 2007 (Statistics Canada, 2010), as recommended by Daneva et al. (2007). If multiple business sectors are applicable, we list them in order of relevance to the project.
- For the dimension *Functional area* (10), we use the following nominal scale: Administration

(Planning), Administration (Organizing), Customer service, Distribution, Finance, Human resources, ICT, Marketing, Sales, Production, and Research & Development (Daft, 2001; Laudon & Laudon, 2000). If multiple business sectors are applicable, we list them in order of relevance to the project.

Project B

1. Project nature	Enhancement and Corrective Maintenance (Project was taken over)
2. Technology gap	Non-existent
3. Level of integration	Stand-alone (initially, but later integrated)
4. Project organization	Developers are co-located, customer is dispersed from developers at national level
5. Availability of project resources	Shared
6. Project relationship structure	Fixed-price (some parts have a time and materials based price)
7. Procurement process	Unknown
8. Market focus	Custom (for the customer), later Mass market (for the customer's customers)
9. Business sector	5611. Office Administrative Services
10. Functional area	Finance
11. Development organization	Very small
12. Size of the client organization	Very small
13. Business risk	Low
14. System risk	Medium
15. Risks to the project	Medium: • Project was taken over
16. Project size	Small
17. Duration	3-9 months (initially)
18. Client experience with IT projects	Medium
19. Methodology used	1. Agile (XP) 2. Low amount of documentation 3. Implied notations are text in natural language + design mock-ups
20. Project governance model	In practice: Manifesto for Agile Software Development

Project C

1. Project nature	Enhancement and Corrective Maintenance (Project was taken over)
2. Technology gap	Non-existent
3. Level of integration	Stand-alone

4. Project organization	Developers are co-located, customer is dispersed from developers at international level
5. Availability of project resources	Some dedicated, some shared
6. Project relationship structure	Fixed-price
7. Procurement process	Business network
8. Market focus	Mass market
9. Business sector	81392. Professional Organizations
10. Functional area	Distribution, Marketing
11. Development organization	Very small
12. Size of the client organization	Very small
13. Business risk	High
14. System risk	Low
15. Risks to the project	High: • Inadequate budget availability • Project was taken over
16. Project size	Small
17. Duration	3-9 months
18. Client experience with IT projects	Low
19. Methodology used	1. Agile (Scrum, XP) 2. Low amount of documentation 3. Implied notations are text in natural language + design mock-ups
20. Project governance model	Manifesto for Agile Software Development

Project D

1. Project nature	Initially Replacement, later Enhancement and Corrective Maintenance
2. Technology gap	Above average
3. Level of integration	Stand-alone (initially, later integrated with various other systems and feeds)
4. Project organization	Developers are co-located, customer is dispersed from developers at national level
5. Availability of project resources	Some dedicated, some shared
6. Project relationship structure	Fixed-price
7. Procurement process	Government, Tender
8. Market focus	Custom (initially only for the customer)
9. Business sector	913. Local, Municipal and Regional Public Administration
10. Functional area	Customer service, Distribution, Marketing
11. Development organization	Very small
12. Size of the client organization	Large
13. Business risk	Medium
14. System risk	Low
15. Risks to the project	Medium: • Technology that is unknown to the project team • Integration with other systems

16. Project size	Large
17. Duration	More than 18 months (including maintenance)
18. Client experience with IT projects	Low
19. Methodology used	1. Official: Agile (Scrum, XP); In practice: in the beginning spiral with agile elements, later agile 2. Medium amount of documentation 3. Implied notations are text in natural language + design mock-ups
20. Project governance model	Manifesto for Agile Software Development

Project F

1. Project nature	Greenfield
2. Technology gap	Above average
3. Level of integration	Integrated (many XML feeds)
4. Project organization	Developers are co-located, customer is dispersed from developers at national level
5. Availability of project resources	Some dedicated, some shared
6. Project relationship structure	Time and materials based price
7. Procurement process	Unknown
8. Market focus	Mass market
9. Business sector	51913. Internet Publishing and Broadcasting, and Web Search Portals
10. Functional area	Distribution, Marketing
11. Development organization	Very small
12. Size of the client organization	Very small
13. Business risk	High
14. System risk	Low
15. Risks to the project	High: • Inadequate budget availability
16. Project size	Small
17. Duration	3-9 months
18. Client experience with IT projects	High
19. Methodology used	1. Agile (Scrum, XP) 2. Low amount of documentation 3. Implied notations are text in natural language + design mock-ups
20. Project governance model	Manifesto for Agile Software Development

Project G

1. Project nature	Enhancement and Corrective Maintenance (Project was taken over)
2. Technology gap	Non-existent
3. Level of integration	Stand-alone

4. Project organization	Developers are co-located, customer is dispersed from developers at international level
5. Availability of project resources	Shared
6. Project relationship structure	Fixed-price
7. Procurement process	Unknown
8. Market focus	Custom (customer)
9. Business sector	54. Professional, Scientific and Technical Services
10. Functional area	Human Resources
11. Development organization	Very small
12. Size of the client organization	Small
13. Business risk	Low
14. System risk	Low
15. Risks to the project	Low
16. Project size	Small
17. Duration	Less than 3 months
18. Client experience with IT projects	Low
19. Methodology used	1. Agile (XP) 2. Low amount of documentation 3. Implied notations are text in natural language
20. Project governance model	In practice: Manifesto for Agile Software Development

Project H

1. Project nature	Greenfield
2. Technology gap	Non-existent
3. Level of integration	Stand-alone
4. Project organization	Developers are co-located, customer is dispersed from developers at national level
5. Availability of project resources	Shared
6. Project relationship structure	Fixed-price
7. Procurement process	Business network
8. Market focus	Mass market
9. Business sector	8122. Funeral Services
10. Functional area	Sales, Marketing
11. Development organization	Very small
12. Size of the client organization	Small
13. Business risk	Low
14. System risk	Low
15. Risks to the project	Low: • IT infrastructure of the customer
16. Project size	Small
17. Duration	Less than 3 months
18. Client experience with IT projects	Low
19. Methodology used	1. Agile (XP) 2. Low amount of documentation 3. Implied notations are text in natural language

20. Project governance model	In practice: Manifesto for Agile Software Development

Project I

1. Project nature	Replacement
2. Technology gap	Non-existent
3. Level of integration	Integrated (Google web services)
4. Project organization	Developers are co-located, customer is dispersed from developers at national level
5. Availability of project resources	Shared
6. Project relationship structure	Fixed-price (subcontractor)
7. Procurement process	Neighborhood
8. Market focus	Custom (initially for the customer itself, but later the customer wanted to sell it as a SaaS application as well)
9. Business sector	44-45. Retail Trade
10. Functional area	Finance
11. Development organization	Very small
12. Size of the client organization	Large
13. Business risk	Medium
14. System risk	Medium
15. Risks to the project	Low
16. Project size	Small
17. Duration	3-9 months
18. Client experience with IT projects	High
19. Methodology used	1. Agile (XP) 2. Medium amount of documentation 3. Implied notations are text in natural language
20. Project governance model	In practice: Manifesto for Agile Software Development

Project J

1. Project nature	Initially Greenfield, later Enhancement and Corrective Maintenance
2. Technology gap	Non-existent
3. Level of integration	Stand-alone
4. Project organization	Developers are co-located, customer is dispersed from developers at national level
5. Availability of project resources	Shared
6. Project relationship structure	Fixed-price
7. Procurement process	Government, Previous experience (Project D)
8. Market focus	Custom (customer)
9. Business sector	913. Local, Municipal and Regional Public Administration

10. Functional area	Customer service, Marketing
11. Development organization	Very small
12. Size of the client organization	Small
13. Business risk	Medium
14. System risk	Low
15. Risks to the project	Low
16. Project size	Small
17. Duration	3-9 months
18. Client experience with IT projects	Low
19. Methodology used	1. Agile (XP) 2. Low amount of documentation 3. Implied notations are text in natural language + design mock-ups
20. Project governance model	In practice: Manifesto for Agile Software Development

Project K

1. Project nature	Greenfield
2. Technology gap	Non-existent
3. Level of integration	Stand-alone
4. Project organization	Co-located
5. Availability of project resources	Some dedicated, some shared
6. Project relationship structure	In-house
7. Procurement process	None (in-house)
8. Market focus	Custom (customer)
9. Business sector	5415. Computer Systems Design and Related Services 518. Data Processing, Hosting, and Related Services
10. Functional area	Finance
11. Development organization	Very small
12. Size of the client organization	Very small
13. Business risk	Medium
14. System risk	Medium
15. Risks to the project	Low
16. Project size	Small
17. Duration	Less than 3 months
18. Client experience with IT projects	Medium
19. Methodology used	1. Agile (Scrum, XP) 2. Low amount of documentation 3. Implied notations are text in natural language + diagrams
20. Project governance model	Manifesto for Agile Software Development

Project L

1. Project nature	Replacement
2. Technology gap	Non-existent
3. Level of integration	Stand-alone
4. Project organization	Developers are co-located, customer is dispersed from developers at national level
5. Availability of project resources	Shared
6. Project relationship structure	Fixed-price
7. Procurement process	Business network, Neighborhood
8. Market focus	Custom (customer)
9. Business sector	5415. Computer Systems Design and Related Services
10. Functional area	Marketing, Sales
11. Development organization	Very small
12. Size of the client organization	Small
13. Business risk	Low
14. System risk	Low
15. Risks to the project	Low
16. Project size	Small
17. Duration	3-9 months
18. Client experience with IT projects	High
19. Methodology used	1. Agile (XP) 2. Low amount of documentation 3. Implied notations are text in natural language + design mock-ups
20. Project governance model	In practice: Manifesto for Agile Software Development

Project M

1. Project nature	Enhancement and Corrective Maintenance (Project was taken over)
2. Technology gap	Non-existent
3. Level of integration	Stand-alone
4. Project organization	Developers are co-located, customer is dispersed from developers at national level
5. Availability of project resources	Shared
6. Project relationship structure	Fixed-price
7. Procurement process	Unknown
8. Market focus	Custom (customer)
9. Business sector	44229. Other Home Furnishings Stores
10. Functional area	Finance
11. Development organization	Very small
12. Size of the client organization	Very small
13. Business risk	Medium
14. System risk	Medium

15. Risks to the project	Medium: • Project was taken over
16. Project size	Small
17. Duration	Less than 3 months
18. Client experience with IT projects	Low
19. Methodology used	1. Agile (XP) 2. Low amount of documentation 3. Implied notations are text in natural language
20. Project governance model	In practice: Manifesto for Agile Software Development

Project N

1. Project nature	Replacement + parts used from previous project
2. Technology gap	Moderate
3. Level of integration	Integrated (3 other applications + FTP)
4. Project organization	Developers are co-located, customer is dispersed from developers at national level
5. Availability of project resources	Shared
6. Project relationship structure	Fixed-price
7. Procurement process	Previous experience (Project L), Neighborhood
8. Market focus	Custom (customer)
9. Business sector	5415. Computer Systems Design and Related Services
10. Functional area	Administration (Planning), Human Resources
11. Development organization	Very small
12. Size of the client organization	Small
13. Business risk	Low
14. System risk	Medium
15. Risks to the project	Medium: • Integration with other systems
16. Project size	Small
17. Duration	Less than 3 months
18. Client experience with IT projects	High
19. Methodology used	1. Agile (XP) 2. Low amount of documentation 3. Implied notations are text in natural language
20. Project governance model	In practice: Manifesto for Agile Software Development

Project O

1. Project nature	Replacement
2. Technology gap	Non-existent
3. Level of integration	Integrated (XML feed)

4. Project organization	Developers are co-located, customer is dispersed from developers at national level
5. Availability of project resources	Shared
6. Project relationship structure	Fixed-price
7. Procurement process	Government, Business network
8. Market focus	Custom (customer)
9. Business sector	913. Local, Municipal and Regional Public Administration
10. Functional area	Customer service, Distribution, Marketing
11. Development organization	Very small
12. Size of the client organization	Large
13. Business risk	Medium
14. System risk	Low
15. Risks to the project	Low: • IT infrastructure of the customer
16. Project size	Small
17. Duration	3-9 months
18. Client experience with IT projects	Low
19. Methodology used	1. Agile (XP) 2. Low amount of documentation 3. Implied notations are text in natural language + design mock-ups
20. Project governance model	In practice: Manifesto for Agile Software Development

Project P

1. Project nature	Enhancement and Corrective Maintenance (Project was taken over)
2. Technology gap	Non-existent
3. Level of integration	Integrated
4. Project organization	Developers are co-located, customer is dispersed from developers at national level
5. Availability of project resources	Shared
6. Project relationship structure	Time and materials based price
7. Procurement process	Unknown
8. Market focus	Custom (customer)
9. Business sector	5151. Radio and Television Broadcasting
10. Functional area	Distribution, Marketing
11. Development organization	Very small
12. Size of the client organization	Large
13. Business risk	Medium
14. System risk	Low
15. Risks to the project	Low
16. Project size	Small
17. Duration	3-9 months
18. Client experience with IT projects	High

19. Methodology used	1. Agile (XP) 2. Low amount of documentation 3. Implied notations are text in natural language
20. Project governance model	In practice: Manifesto for Agile Software Development

Project R

1. Project nature	Greenfield
2. Technology gap	Above average
3. Level of integration	Integrated (various server scripts)
4. Project organization	Co-located
5. Availability of project resources	Some dedicated, some shared
6. Project relationship structure	In-house
7. Procurement process	None (in-house)
8. Market focus	Mass market
9. Business sector	518. Data Processing, Hosting, and Related Services 5415. Computer Systems Design and Related Services
10. Functional area	Sales, Finance, Production
11. Development organization	Very small
12. Size of the client organization	Very small
13. Business risk	Medium
14. System risk	Medium
15. Risks to the project	Medium: • Integration with various server scripts
16. Project size	Medium
17. Duration	3-9 months
18. Client experience with IT projects	Medium
19. Methodology used	1. Agile (Scrum, XP) 2. Low amount of documentation 3. Implied notations are text in natural language + diagrams
20. Project governance model	Manifesto for Agile Software Development

Project S

1. Project nature	Greenfield
2. Technology gap	Non-existent
3. Level of integration	Stand-alone
4. Project organization	Developers are co-located, customer is dispersed from developers at national level
5. Availability of project resources	Shared
6. Project relationship structure	Fixed-price
7. Procurement process	Government, Previous experience (Project D)

8. Market focus	Custom (customer)
9. Business sector	913. Local, Municipal and Regional Public Administration
10. Functional area	Marketing
11. Development organization	Very small
12. Size of the client organization	Large
13. Business risk	Low
14. System risk	Low
15. Risks to the project	Low
16. Project size	Small
17. Duration	Less than 3 months
18. Client experience with IT projects	Low
19. Methodology used	1. Agile (XP) 2. Low amount of documentation 3. Implied notations are text in natural language + design mock-ups
20. Project governance model	In practice: Manifesto for Agile Software Development

Project T

1. Project nature	Greenfield
2. Technology gap	Non-existent
3. Level of integration	Stand-alone
4. Project organization	Developers are co-located, customer is dispersed from developers at national level
5. Availability of project resources	Shared
6. Project relationship structure	Fixed-price
7. Procurement process	Business network
8. Market focus	Custom (customer)
9. Business sector	611. Educational Services
10. Functional area	Marketing
11. Development organization	Very small
12. Size of the client organization	Large
13. Business risk	Low
14. System risk	Low
15. Risks to the project	Low: • IT infrastructure of the customer
16. Project size	Small
17. Duration	Less than 3 months
18. Client experience with IT projects	Low
19. Methodology used	1. Agile (XP) 2. Low amount of documentation 3. Implied notations are text in natural language
20. Project governance model	In practice: Manifesto for Agile Software Development

Project U

1. Project nature	Enhancement and Corrective Maintenance (Project was taken over)
2. Technology gap	Moderate
3. Level of integration	Integrated (various XML feeds)
4. Project organization	Developers are co-located, customer is dispersed from developers at national level
5. Availability of project resources	Shared
6. Project relationship structure	Time and materials based price
7. Procurement process	Previous experience (Project P)
8. Market focus	Mass market
9. Business sector	5151. Radio and Television Broadcasting
10. Functional area	Distribution
11. Development organization	Very small
12. Size of the client organization	Large
13. Business risk	Low
14. System risk	Low
15. Risks to the project	Medium: • Integration with various XML feeds • Project was taken over
16. Project size	Small
17. Duration	3-9 months
18. Client experience with IT projects	High
19. Methodology used	1. Agile (XP) 2. Low amount of documentation 3. Implied notations are text in natural language
20. Project governance model	In practice: Manifesto for Agile Software Development

Project V

1. Project nature	Greenfield
2. Technology gap	Non-existent
3. Level of integration	Stand-alone
4. Project organization	Developers are co-located, project manager and customer are dispersed from developers and each other at national level
5. Availability of project resources	Shared
6. Project relationship structure	Fixed-price
7. Procurement process	Previous experience (Hosting)
8. Market focus	Custom (multiple parties)
9. Business sector	91. Public Administration 23. Construction
10. Functional area	Administration (Organizing)

11. Development organization	Very small
12. Size of the client organization	Large
13. Business risk	Medium
14. System risk	Low
15. Risks to the project	Medium: • Many different parties involved
16. Project size	Large
17. Duration	3-9 months
18. Client experience with IT projects	High
19. Methodology used	1. Waterfall with Agile (XP) elements 2. High amount of documentation 3. Implied notations are text in natural language + diagrams
20. Project governance model	Unknown

Project W

1. Project nature	Enhancement and Corrective Maintenance (Project was taken over)
2. Technology gap	Moderate
3. Level of integration	Integrated (with 2 web services)
4. Project organization	Developers are co-located, customer is dispersed from developers at international level
5. Availability of project resources	Shared
6. Project relationship structure	Fixed-price
7. Procurement process	Unknown
8. Market focus	Mass market
9. Business sector	8133. Social Advocacy Organizations
10. Functional area	Sales, Marketing, Distribution
11. Development organization	Very small
12. Size of the client organization	Small
13. Business risk	Medium
14. System risk	Low
15. Risks to the project	Medium: • Project was taken over
16. Project size	Small
17. Duration	Less than 3 months
18. Client experience with IT projects	Medium
19. Methodology used	1. Agile (XP) 2. Low amount of documentation 3. Implied notations are text in natural language + design mock-ups
20. Project governance model	In practice: Manifesto for Agile Software Development

Project X

1. Project nature	Greenfield
2. Technology gap	Moderate
3. Level of integration	Integrated (various web services)
4. Project organization	Developers are co-located, customer is dispersed from developers at national level
5. Availability of project resources	Shared
6. Project relationship structure	Time and materials based price
7. Procurement process	Business network
8. Market focus	Mass market
9. Business sector	518. Data Processing, Hosting, and Related Services
10. Functional area	Sales, Finance, Production
11. Development organization	Very small
12. Size of the client organization	Small
13. Business risk	Medium
14. System risk	Medium
15. Risks to the project	Medium: • Integration with various web services
16. Project size	Medium
17. Duration	3-9 months
18. Client experience with IT projects	Medium
19. Methodology used	1. Agile (XP) 2. Low amount of documentation 3. Implied notations are text in natural language
20. Project governance model	In practice: Manifesto for Agile Software Development

Project Y

1. Project nature	Greenfield
2. Technology gap	Moderate
3. Level of integration	Integrated (various web services)
4. Project organization	Developers are co-located, customer is dispersed from developers at national level
5. Availability of project resources	Some dedicated, some shared
6. Project relationship structure	Fixed-price (subcontractor)
7. Procurement process	Unknown
8. Market focus	Mass market
9. Business sector	519. Other Information Services
10. Functional area	Sales
11. Development organization	Very small
12. Size of the client organization	Large
13. Business risk	Medium
14. System risk	Low
15. Risks to the project	Medium: • Integration with various web services

16. Project size	Small
17. Duration	More than 18 months
18. Client experience with IT projects	High
19. Methodology used	1. Agile (XP) 2. Medium amount of documentation 3. Implied notations are text in natural language + diagrams
20. Project governance model	In practice: Manifesto for Agile Software Development

Project Z

1. Project nature	Greenfield
2. Technology gap	Moderate
3. Level of integration	Integrated (various web services)
4. Project organization	Developers are co-located, customer is dispersed from developers at national level
5. Availability of project resources	Some dedicated, some shared
6. Project relationship structure	Fixed-price (subcontractor)
7. Procurement process	Previous experience (Project Y)
8. Market focus	Custom (customer)
9. Business sector	519. Other Information Services
10. Functional area	Distribution
11. Development organization	Very small
12. Size of the client organization	Small
13. Business risk	Low
14. System risk	Low
15. Risks to the project	Low
16. Project size	Small
17. Duration	More than 18 months
18. Client experience with IT projects	High
19. Methodology used	1. Agile (XP) 2. Medium amount of documentation 3. Implied notations are text in natural language + diagrams
20. Project governance model	In practice: Manifesto for Agile Software Development

Appendix B: Problem bundle traceability matrix

Below we give a traceability matrix for the relations in the problem bundle (Figure 14).

The numbers before each problem are used to trace the relations between the problems back to the data we collected; so the relations are identified by stating the causality between the problems. For example 1. → 2. means the relation from problem 1. to problem 2.

In the data collection column we give the data sources where the relation is based on. This can be a project (chapter 4 and more specifically Appendix A: Project dimensions), a document (chapter 5), an interview (chapter 6), observations we made, our own experience in the company, or logical deduction.

Relation	**Data sources**
1. → 2.	Interview Manager C
2. → 3.	Interview Manager C
2. → 5.	Interview Manager C; own experience
3. → 4.	Interview Manager C; own experience
4. → 6.	Own experience
5. → 6.	Own experience
6. → 18.	E-mail conversation with Manager C
7. → 8.	Interview Manager B
7. → 9.	Project B, Project C, Project M, Project U, Project W; Interview Manager B; own experience
9. → 18.	Project B, Project C, Project M, Project U, Project W; Interview Manager B; own experience; logical deduction
10. → 11.	Interview Manager C; own experience
10. → 12.	Logical deduction (Manager C gives 10. → 17. in the e-mail conversation)
12. → 17.	E-mail conversation with Manager C
13. → 14.	Own experience
14. → 17.	E-mail conversation with Manager C
15. → 17.	E-mail conversation with Manager C; own experience
16. → 17.	Own experience
17. → 18.	E-mail conversation with Manager C
18. → 19.	Logical deduction (Manager C gives 18. → 107. in the e-mail conversation)
19. → 20.	Observations; own experience
20. → 41.	Observations; own experience
20. → 105.	Own experience; observations
20. → 107.	Logical deduction (Manager C gives 18. → 107. in the e-mail conversation)
20. → 110.	Observations
20. → 112.	Interview Manager C; observations; own experience

21. → 22.	Interview Developer A, interview Manager B, interview Manager C
22. → 26.	Interview Manager C; Project D
23. → 25.	Interview Manager C
24. → 25.	Interview Manager C
25. → 26.	Interview Manager C
26. → 34.	Observations
27. → 28.	Observations; own experience
28. → 29.	Interview Manager B, interview Manager C
28. → 30.	Observations
28. → 31.	Documents: SLAs; interview Manager B
28. → 32.	Documents: SLAs & quotes
29. → 34.	Observations
30. → 42.	Documents: SLAs; interview Manager B
30. → 43.	Interview Manager B
31. → 47.	Interview Manager B, interview Manager C
32. → 66.	Interview Developer A, interview Manager B, interview Manager C; observations
33. → 35.	Observations
34. → 35.	Observations
34. → 36.	Interview Manager C
35. → 37.	Logical deduction
36. → 47.	Interview Developer A, interview Manager B, interview Manager C
37. → 38.	Interview Manager C
37. → 39.	Interview Developer A; own experience
37. → 40.	Interview Manager C
37. → 41.	Interview Manager C
40. → 59.	Observations
41. → 59.	Observations
44. → 46.	Interview Developer A
45. → 47.	Interview Developer A, interview Manager B, interview Manager C
46. → 47.	Logical deduction
47. → 48.	Observations
48. → 49.	Logical deduction
50. → 51.	Interview Manager B
50. → 52.	Interview Manager B
52. → 53.	Interview Manager B
52. → 54.	Logical deduction
53. → 55.	Interview Manager B
54. → 82.	Interview Manager B
55. → 57.	Interview Manager B
55. → 59.	Logical deduction
56. → 57.	Interview Manager B
57. → 59.	Interview Manager B
58. → 59.	Interview Manager B, interview Manager C
60. → 61.	Interview Developer A, interview Manager B, interview Manager C
61. → 63.	Interview Developer A, interview Manager B, interview Manager C
61. → 64.	Observations; own experience
62. → 63.	Interview Manager B
63. → 65.	Interview Manager B, interview Manager C
64. → 65.	Observations; own experience

65. → 66.	Interview Developer A, interview Manager B, interview Manager C; observations; own experience
66. → 67.	Interview Developer A
67. → 77.	Interview Developer A
68. → 69.	Interview Manager B
68. → 70.	Interview Manager C
69. → 71.	Interview Developer A
70. → 69.	Interview Manager B
70. → 72.	Interview Manager C
71. → 74.	Interview Developer A
73. → 74.	Interview Developer A
74. → 77.	Interview Developer A
75. → 76.	Interview Manager B, interview Manager C
75. → 78.	Interview Manager B
76. → 77.	Interview Developer A
78. → 80.	Interview Manager B; logical deduction
79. → 80.	Interview Developer A, interview Manager B, interview Manager C
81. → 82.	Interview Manager B
82. → 83.	Interview Manager B
84. → 85.	Interview Developer A, interview Manager B, interview Manager C
85. → 86.	Interview Developer A, interview Manager B, interview Manager C
85. → 89.	Interview Manager B
87. → 88.	Interview Manager B, interview Manager C
88. → 89.	Interview Manager B, interview Manager C
88. → 91.	Interview Manager B, interview Manager C
89. → 90.	Logical deduction
90. → 91.	Interview Manager B, interview Manager C
91. → 93.	Observations
91. → 99.	Interview Manager B
92. → 93.	Observations
94. → 95.	Documents: quotes; observations
95. → 97.	Interview Manager B
95. → 98.	Interview Manager B
96. → 97.	Observations
97. → 100.	Logical deduction
98. → 99.	Interview Manager B, interview Manager C
98. → 100.	Logical deduction
100. → 101.	Logical deduction
100. → 102.	Interview Manager C (Manager B gives 97. → 102. and 98. → 102.)
102. → 103.	Interview Manager C
103. → 110.	Interview Manager B
104. → 110.	Interview Manager B
106. → 107.	E-mail conversation with Manager C
107. → 108.	Observations
107. → 111.	Logical deduction
108. → 109.	Logical deduction
109. → 110.	Logical deduction; own experience
110. → 118.	Observations
111. → 112.	Observations
112. → 114.	Interview Manager C

113. → 114.	Interview Developer A, interview Manager B, interview Manager C; own experience
114. → 115.	Interview Developer A, interview Manager C
115. → 116.	Observations
115. → 117.	Observations
117. → 118.	Logical deduction
118. → 119.	Logical deduction