DOCUMENTING FOR PROGRAM COMPREHENSION IN AGILE SOFTWARE DEVELOPMENT

Master's (one year) thesis in Informatics (15 credits)

Fabian Kiss

Autumn 2010:MI11
Title: Documenting for Program Comprehension in Agile Software Development

Year: 2011

Author: Fabian Kiss

Supervisor: Mikael Lind

Abstract:
Program comprehension, i.e. to understand from its source code what a computer program does, is crucial for change and maintenance in software development. In this thesis, it is looked for innovative documentation techniques and tools that support program comprehension, but that are also conform to agile values and principles – commonly, documentation is considered critical due to the agile value “working software over comprehensive documentation.”¹ First, a research framework is developed that embodies detailed requisites for such techniques and tools. Apart from its internal use for examining techniques and tools subsequently obtained from a literature search, this framework is intended to be likewise employed by software practitioners. Eventually, the findings of a series of survey studies conducted in an industrial software organization for the primary purpose of evaluating the obtained techniques and tools are analyzed. Three innovative techniques that meet all requisites are revealed. These are regarded by practitioners independently from the support of program comprehension as helpful for a change impact analysis conducted by non-developers. Therefore, a requisite deduced from the highest priority in agile software development – customer satisfaction – is met. It says that a technique or tool has to directly induce a benefit for non-developer stakeholders besides the benefits for them which are indirectly induced by the support of program comprehension, e.g. a potentially improved source code quality. Further, the technique most beneficial for developers as well as for non-developers among the three techniques is identified, which bases on design rationales – textual information related to the source code that states the reasons why a part of the program has been implemented in a certain way. Secondarily, the studies revealed that the research framework is difficult to understand for practitioners due to its unstructured form.

Keywords:
Software documentation, Source code documentation, Program comprehension, Program understanding, Agile software development

¹Manifesto for Agile Software Development (http://agilemanifesto.org/)
# Contents

List of Tables .................................................. VI
List of Figures .................................................. VII
List of Listings .................................................. VIII

1 Introduction ................................................. 1
   1.1 Previous research in program comprehension and agile software development .... 1
   1.2 Rationale for supporting program comprehension by documentation ............... 2
   1.3 Research purpose and research goals ................................................... 3
   1.4 Research questions ............................................................................ 4
   1.5 Delimitations ..................................................................................... 5
   1.6 Research approach .......................................................................... 5
   1.7 Background of the author .................................................................. 6
   1.8 Overview of the structure of the thesis ............................................. 6

2 Background knowledge ......................................... 8
   2.1 Program comprehension .................................................................. 8
   2.2 Agile software development ............................................................ 9
      2.2.1 Agile values .............................................................................. 9
      2.2.2 Agile principles ...................................................................... 10
      2.2.3 Extreme Programming (XP) .................................................... 11
      2.2.4 Agile practices ...................................................................... 15
      2.2.5 Scrum .................................................................................... 15

3 Research framework ............................................ 17
   3.1 The LowLevelContext requisite ...................................................... 17
   3.2 The HighLevelBenefit requisite ...................................................... 17
   3.3 The NoSeparateArtifact requisite ................................................... 19
   3.4 The PrimarilyProgramComprehension requisite ............................... 20
   3.5 The ArbitraryAgileMethodology requisite ....................................... 21
   3.6 The ArbitraryProgrammingLanguage requisite .................................. 22

4 Literature review ................................................. 24
   4.1 Search method ............................................................................... 24
   4.2 Inspection results .......................................................................... 27
   4.3 Literate Programming ................................................................... 29
      4.3.1 Summary .............................................................................. 29
      4.3.2 Discussion ............................................................................ 30
   4.4 Doc comments .............................................................................. 33
      4.4.1 Summary .............................................................................. 33
      4.4.2 Discussion ............................................................................ 37
   4.5 Unit tests ....................................................................................... 39
      4.5.1 Summary .............................................................................. 39
      4.5.2 Discussion ............................................................................ 39
<table>
<thead>
<tr>
<th>Appendix A: Unrefactored unit test</th>
<th>109</th>
</tr>
</thead>
<tbody>
<tr>
<td>Appendix B: Unabstracted scenario diagram</td>
<td>110</td>
</tr>
<tr>
<td>Appendix C: Meet/violate table for each technique or tool</td>
<td>111</td>
</tr>
<tr>
<td>Appendix D: Patterns for Agile Documentation</td>
<td>111</td>
</tr>
<tr>
<td>Appendix E: Questionnaires</td>
<td>113</td>
</tr>
<tr>
<td>Appendix F: Likert-scale questions results</td>
<td>125</td>
</tr>
</tbody>
</table>
# List of Tables

1. Narrowing down sequence for research concepts ........................................ 25
2. Meet / violate table for each technique or tool ........................................... 111
3. *Pattern Thumbnails* for Agile Documentation ............................................. 112
4. Results of the Likert-scale questions from questionnaires #2 to #9 ............... 125
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Example of a user story</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td>Venn diagram with research concepts</td>
<td>25</td>
</tr>
<tr>
<td>3</td>
<td>Beginning of the explanation extracted as TeX part from the literate program (listing 2) and subsequently processed for print output</td>
<td>32</td>
</tr>
<tr>
<td>4</td>
<td>JavaDoc-generated HTML documentation of Java class java.net.CacheResponse</td>
<td>36</td>
</tr>
<tr>
<td>5</td>
<td>Scenario diagram of a test case for JPacman</td>
<td>42</td>
</tr>
<tr>
<td>6</td>
<td>Visualizations of links from source code to user stories in Zelda</td>
<td>48</td>
</tr>
<tr>
<td>7</td>
<td>Representations of a design rationale</td>
<td>52</td>
</tr>
<tr>
<td>8</td>
<td>Example of a slide used in the presentation for the conduct of a study</td>
<td>65</td>
</tr>
<tr>
<td>9</td>
<td>Fully labeled Likert scale with five response categories as used in the questionnaires</td>
<td>65</td>
</tr>
<tr>
<td>10</td>
<td>Histograms of the participants’ work experience as a software developer, project manager, or product manager</td>
<td>70</td>
</tr>
<tr>
<td>11</td>
<td>Pie chart of the participants’ education that is directly related to their positions as a software developer, project manager, or product manager (question #1.5)</td>
<td>70</td>
</tr>
<tr>
<td>12</td>
<td>Stacked bar charts in respect of the thesis’ topic in general (questionnaire #2)</td>
<td>71</td>
</tr>
<tr>
<td>13</td>
<td>Stacked bar charts in respect of the requisites (questionnaire #3)</td>
<td>72</td>
</tr>
<tr>
<td>14</td>
<td>Sequence diagram of a login use case</td>
<td>76</td>
</tr>
<tr>
<td>15</td>
<td>Stacked bar charts in respect of the understandability of sequence diagrams (questionnaire #4)</td>
<td>78</td>
</tr>
<tr>
<td>16</td>
<td>Stacked bar charts in respect of a change impact analysis based on sequence diagrams (questionnaire #5)</td>
<td>79</td>
</tr>
<tr>
<td>17</td>
<td>Explorer view of Eclipse as modified by the tool of Ratanotayanon et al. (2009)</td>
<td>81</td>
</tr>
<tr>
<td>18</td>
<td>Understandability of the three presented visualizations of links from source code to user stories (question #6.2)</td>
<td>83</td>
</tr>
<tr>
<td>19</td>
<td>Stacked bar charts in respect of a change impact analysis based on visualizations of links from source code to user stories (questionnaire #7)</td>
<td>84</td>
</tr>
<tr>
<td>20</td>
<td>Understandability of the two examples of a design rationale (questionnaire #8.2)</td>
<td>106</td>
</tr>
<tr>
<td>21</td>
<td>Stacked bar charts in respect of a change impact analysis based on design rationales (questionnaire #9)</td>
<td>87</td>
</tr>
<tr>
<td>22</td>
<td>Unabstracted scenario diagram of a test case for JPacman opposed to the diagram in figure 5</td>
<td>110</td>
</tr>
<tr>
<td>List of Listings</td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------------</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>A unit test written with the <em>JUnit</em> framework (excerpt of a Java class)</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>Beginning of a literate program that implements an algorithm for printing the first 1000 prime numbers</td>
<td>31</td>
</tr>
<tr>
<td>3</td>
<td>Complete Pascal source code extracted from the literate program shown in listing 2</td>
<td>31</td>
</tr>
<tr>
<td>4</td>
<td>Java source code of class <code>java.net.CacheResponse</code></td>
<td>35</td>
</tr>
<tr>
<td>5</td>
<td>The original – unrefactored – version of the unit test from listing 1</td>
<td>109</td>
</tr>
</tbody>
</table>
1 Introduction

Evidently, maintenance and further development of existing software are major tasks of most software developers. This is particularly true for computer programs that are coined by frequent changes to their source code. (Pigoski 1996, Seacord et al. 2003) Therefore, program comprehension – to understand from its source code what a computer program does (Brooks 1983) – is a highly crucial aspect of software development. Agile software development methodologies acknowledge this aspect by striving for high-quality source code (Martin 2009) with prominent techniques like, for example, source code refactoring (Fowler 2000). With those, the readability of the source code is enhanced, what obviously benefits program comprehension. Practiced continuously – as recommended – source code refactoring can unfortunately also potentiate the necessity of program comprehension, because the changed source code has to be understood once again (van Deursen 2001).

Besides that, understanding a program in terms of program comprehension is not solely facilitated by such techniques. Especially the program knowledge – needed for abstracting from the code – is further relevant for that. The likely decay of that knowledge among the programmers on a long-term basis is a ubiquitous problem. The traditional remedy for that is practicing documentation. (de Souza et al. 2007) When referring to the Agile Manifesto (Beck et al. 2001a), abandoning any documentation is widespread due to a mistaken interpretation of the agile value “working software over comprehensive documentation” as a ban of documentation at all. However, documentation can also be potentially beneficial from an agile perspective without any universal contradiction. (Selic 2009) In fact, a documentation rooted in the low-level nature of source code, which is suggesting for the ambition for a better program comprehension, is largely conform to a vast part of agile practices (Ratanotayanon et al. 2009). For example, unit tests developed as part of the common agile testing practice (Martin 2003) could, besides their testing purpose, also be the base of an executable work product with a documenting purpose (Ambler 2010a, Forward & Lethbridge 2002b, van Deursen 2001). In addition, this low-level documenting approach could serve to elicit a documentation that supports the understanding on a system-level apart from the initial purpose of supporting program comprehension. (Tilley 1996, 2009). Such a resulting documentation could be devoted to other stakeholders than the developers, for example, product managers, and their further abstracted understanding of the program in development. Or it could be a part of the completed software product itself, e.g. a reference for an Application Programming Interface (API).

1.1 Previous research in program comprehension and agile software development

Essential for relating the research done for this thesis to previous research in appropriate research fields is the elaboration of the lack of research in program comprehension and agile software development as a combined topic. For such an elaboration it is advisable to take into account documentation as a connecting topic: In the research community, documentation is a topic that appears often in context with the topic agile software development, the same applies to documentation in context with program comprehension. In the first case, documentation is considered rather as an opponent, in the second case rather as a companion. There is virtually
no likewise popular relation between the topics agile software development and program comprehension, what makes this thesis largely unique. The following reasons might be responsible for that:

- Agile software development and program comprehension are both relatively specific topics with no evident intersection. While agile software development does not explicitly address such foundational issues like program comprehension, that does not explicitly address a particular software development methodology. This is a typical case of the well-known gap between issues related to programming languages and issues related to software engineering despite the interweave of both (Maletic & Kagdi 2008).

- Program comprehension is mainly known as a matter of cognition and thus predestined to be of rather academic interest, e.g. in cognitive science. In contrast to that, agile software development is a matter of practice, often mentioned in connection with “software craftsmanship” (Martin 2009). This theory-practice dichotomy impedes an access to these topics in combination.

- Agile software development emerged noticeably in the late 1990s (Beck 1999, Schwaber & Beedle 2002), was manifested – literally – with the Agile Manifesto (Beck et al. 2001a) in 2001, and became popular in the recently elapsed decade. Contrary to that, program comprehension was coined as a term already in the late 1970s (Love 1977, Brooks 1978) but apparently did not evoke a similar increase of its popularity meanwhile. Thus, it might not be suggesting to combine such an “obsolete” topic with a “hot” topic like agile software development.

However, program comprehension in agile software development has not been completely neglected by the research community. The following statements give a justification to do further research in this respect:

“The need for effective tools to support program comprehension is particularly high in Agile, where the software written today is next week’s legacy code.” – (Ratanotayanon et al. 2009, sec. 1)

“More theories about the collaborative nature of program comprehension [...] are needed.” – (Storey 2005, sec. 6.2)

1.2 Rationale for supporting program comprehension by documentation

So far, the significance of program comprehension in agile software development was elaborated. Further, it was indicated that agile software development does not support that sufficiently in practice. Additionally using documentation for supporting program comprehension could remedy this deficiency:

- As mentioned, program comprehension is designated to be supported more or less by a high code quality in agile software development (Martin 2009). In this particular context, the need for further documentation should be expendable to a large part, following
the idea “the code is the documentation” (Selic 2009). But neither a documentation for the sole purpose of supporting program comprehension nor one for another purpose can be constituted comprehensively by that. As said before, on a long-term basis a decay of knowledge that is necessary for a certain degree of program comprehension is very likely (de Souza et al. 2007). That knowledge could be program knowledge, i.e. knowing how the program works, as well as domain knowledge to relate that program knowledge usefully to its application domain and vice versa. The decay itself could probably occur when a team of software developers changes frequently or when one of them is not available (anymore), for example, because the one developer left the software organization.

• Besides this dynamic knowledge decay, different states of knowledge among developers of a software organization could – as a more static emergence – likewise necessitate a documentation beyond the idea “the code is the documentation”. A very common scenario for that is the employment of an experienced team for the initial development of a program and the employment of a less experienced team, e.g. trainees, for its maintenance (Selic 2009). Those certainly face considerable challenges in respect of program comprehension as it is to assume that they have been employed recently.

• Achieving a certain level of code quality might be limited due to technical reasons or a specific software architecture. For example, the compiler for the used programming language could require special accommodations to the code which are counterproductive regarding a documentation that is, like here, not devoted purely to a low-level context (Selic 2009). In the worst case, the “documentation” constituted by such an accommodated code is more obfuscating than its absence would be. A concrete scenario for that could be the challenges of multithreaded programming in respect of concurrency (Ottoni et al. 2006).

• In agile software development, face-to-face communication is preferred to any form of documentation for conveying information, which will become clear when dealing with the agile principles in section 2.2.2. However, a face-to-face communication is not superior in every case. It is possible that nobody (anymore) knows any crucial information to convey. Then, the software developers have to “consult” the source code and program comprehension is inevitable. Therefore, a documentation supporting program comprehension would be beneficial anyway.

1.3 Research purpose and research goals

After the motivation for the topic of this thesis, the research purpose of it is to address: to show that and how documenting particularly for the support of program comprehension is feasible without impeding the agility demanded by agile software development. For that, documentation techniques and tools are considered as the research results of the thesis. In this context, tools are seen as implementations of the techniques according to the notion of Ghezzi et al. (1991). As there obviously is an abundance of documentation techniques and tools, it is looked for existing techniques and tools instead of developing new ones. With them, the following research goals are associated:

• Of course, the techniques and tools are related to program comprehension in a supportive manner.
Further, they are suitable for any software organization that develops software products with an agile software development methodology. Thus, they have to be suitable for any agile software development methodology and any further software development framework. This is essential for the generality of the results of the research done for this thesis.

Preferably, the techniques and tools excel in being innovative. Even though the topic of the thesis is obviously narrow enough to provide originality, this further ensures the novelty of it.

A benefit for the customer of a software product or a representative like, for example, a product manager, is induced directly by the techniques and tools such that meeting the customer satisfaction is not neglected, which is defined as the highest priority in agile software development (Beck et al. 2001b). Relying on an indirectly induced benefit for the customer from a documentation that is solely related to program comprehension might be insufficient in this respect as program comprehension itself does not serve customer satisfaction. In contrast to that, the above-mentioned understanding on a system-level as an additional benefit, with non-developers as a target audience, seems to be capable of directly inducing a benefit in respect of customer satisfaction. Hence, the emergence of such an additional benefit is set as a requirement for the techniques and tools.

### 1.4 Research questions

The leading research question of this thesis is which concrete techniques and tools meet the above presented research goals or, more generally, whether and to which extent those can be met. If they can not be met at all, or not fully, then does that apply to a particular technique or tool only? Why a technique or tool does not meet a goal? And is there a possibility to modify either the goal or the technique / tool such that the goal is met? How to accomplish that?

Not surprisingly, it is evident that answering these questions universally and comprehensively can hardly be done. Especially the lastly presented goal would require long-term empirical studies over years in many software organizations for that, which certainly would exceed a thesis work. Because of this situation, the research was conducted to a certain extent with an exploratory approach in the theoretical part as well as in the empirical part of this thesis. Particularly, no concrete instantiation of the mentioned directly induced benefit regarding the lastly presented goal was anticipated. Indeed, it is a further research question what such a benefit could be. Another research question that is supposed to be answered in this context is whether there can be found any technique or tool which is particularly promising in respect of meeting the research goals compared with other found techniques and tools?

Moreover, a research framework was developed beforehand, which further narrowed the view on the topic by specifying requirements and delimitations of the research goals. However, it is not included in this introduction. Due to its novelty and its detailedness, this framework has been considered as a part of the research itself – in the form of an additional research result, apart from the techniques or tools that had to meet it. Thus, it can potentially be used by practitioners for evaluating given techniques or tools according to the research purpose of this thesis. Therefore, the applicability of it in practice is a secondary research question: Do practitioners appraise it as justified in agile software development? And do they appraise it as important for their personal involvement in that?


1.5 Delimitations

Even though several delimitations of the conducted research are specified by the research framework, the following delimitations have to be stated here for clarification:

- Contrary to techniques or tools, processes are not considered as objects of investigation. Certainly, they differ too much from organization to organization in order to comply with an adequate generality.

- According to the cited notion of a tool of Ghezzi et al. (1991), a single tool is an implementation of multiple techniques. Therefore, only the fundamental technique implemented by a tool is considered such that both terms could largely be used analogously.

- It is assumed that a documentation is produced and maintained solely by the programmers. Investigating program comprehension implicates this assumption.

- For the same reason, a software product is to treat reduced to the program that it comprises – regarding the equivocality “programs vs. software products” (Mall 2003, p. 25),

- Similarly, the equivocality “developer vs. programmer” (Ionas 2004) can be resolved by the designation of a software developer who programs – where applicable – as a programmer. Thus, both terms refer to the same person.

1.6 Research approach

For providing an outline of the research that was done for this thesis, essentially to mention is that it comprised two parts – a theoretical part followed by an empirical part. As the topic of the thesis is of practical relevance, relying solely on theory-based reasoning was seen as deficient and thus empirical research in practice as indispensable. Overall, both parts complied with a qualitative research perspective even though also quantitative data was collected in the empirical part. Important to remark is that both parts were performed consecutively: The results of the theoretical part were intended to be the object of research in the empirical part.

The theoretical part of the research was carried out by an extensive literature review. Research works about techniques and tools that were considered as potential research results were obtained by systematic keyword searches. These searches were embedded in a search method which stepwise combined the self-contained subjects addressed by the topic of the thesis – program comprehension, documentation, and agile software development. Then the techniques and tools were discussed along the question whether and to which extent they meet the research framework.

Subsequently to the theoretical part, the empirical part was carried out by a series of survey studies. From the techniques and tools that were obtained through the literature review, the promising and innovative ones were selected to be studied empirically – each with a single study. Essentially, such a study was conducted by surveying software developers, project managers, and product managers of a particular software organization – HolidayCheck AG – which develops software products with an agile software development methodology. Primarily, the
evaluation of the selected techniques and tools by them was of interest. In this evaluation, the research questions about the techniques and tools which the empirical part has not answered were investigated. Secondarily, the research framework itself was evaluated with one of the studies to investigate its applicability by practitioners.

Before both parts, the research framework was developed along deductions from the research goals and further criteria such as, for example, general issues in agile software development. This was done by formulating in detail requisites for the techniques and tools, which include a rationale and bear a meaningful name. As said, the framework entirely embraced the conducted research – the theoretical part as well as the empirical part. Contrary to the framework, each of the two parts has been considered rather as separate research in terms of the research strategy and research methods used in that part. As said, both parts were performed consecutively. Thus, the particular research design is not expounded here but in the appropriate presentation of each part (see section 1.8). Nonetheless, it should be mentioned at this place that the conducted research in general followed a scientific perspective coined by the philosophical stance of pragmatism, which can be characterized by the following attributes:

“Pragmatism acknowledges that all knowledge is approximate and incomplete and [...] truth is uncovered in the process of rational discourse, and is judged by the participants as whatever has the better arguments. [...] Pragmatists use any available methods, and strongly prefer mixed methods research, where several methods are used to shed light on the issue under study.” – (Easterbrook et al. 2008, p. 292)

According to the outline of the conducted research here and the presentation of the research questions above, the actually applied scientific perspective complied with these attributes. Indeed, following a pragmatic perspective was done intentionally as the context of this thesis – agile software development – incorporates a likewise pragmatic perspective: Its theoretical roots lies more in pragmatism than in positivism (Nerur & Balijepally 2007).

1.7 Background of the author

The idea for this thesis emerged from the previous professional experience of the author, who has been working as a software developer and who is familiar with agile software development in practice. Thus, the above exposed motivation could be confirmed by practical experience. Furthermore, this experience was particularly helpful for conducting the survey studies. However, another implication might by an adherence of the thesis to the perspective of a software developer. Therefore, its readers should own a minimum knowledge of contemporary software development practices.

1.8 Overview of the structure of the thesis

While the structure of each section of the thesis will be outlined at the beginning of that, an overview of the overall structure is given here: After this introduction, section 2 will provide the necessary background knowledge for understanding the thesis in general as well as for classifying it. Subsequently, the research framework will be dealt with in section 3. Both
research parts embraced by that will be handled in section 4 (literature review) and section 5 (survey studies). As clarified, the latter will base on the findings of the former. Apart from that, it was tried to keep each of the two parts maximally self-contained in favor of a better readability. Consequently, the discussion of each part is largely included in the presentation of that part itself. Eventually, section 6 will conclude the thesis by relating the research results to the initial research questions and by reflecting on the research results as well as on the research process according to scientific criteria. Furthermore, suggestions for using the results for future research will be remarked.
2 Background knowledge

This section deals with the topics addressed in the thesis – program comprehension (section 2.1) and agile software development (section 2.2). From each topic, only the fundamentals are covered, which are essential for understanding that topic in the context of the thesis. Furthermore, apparently common but vague definitions are clarified. Documentation in software development is not dealt with in a separate topic in this section, because most of its relevant aspects have already been clarified above in the introduction. At this place, it only has to be remarked that the term documentation is used as a synonymous short form of software documentation. Likewise to documentation as a topic, basic knowledge about software development in general is presumed known.

2.1 Program comprehension

The notion of program comprehension originates from processes for maintaining and enhancing computer programs (Littman et al. 1987). It simply means to understand a completed computer program (Brooks 1983). Because of that, program understanding is used as a synonym for it, what becomes evident when doing a keyword search with a research database like, for example, IEEE (2010). Clearly, not a possibly compiled executable form but the source code is considered as a computer program in this context. Nonetheless, it is not clear what it means to understand a program. For this thesis the definition by Biggerstaff et al. (1993) is used:

“A person understands a program when [...] able to explain the program, its structure, its behavior, its effects on its operational context, and its relationships to its application domain in terms that are qualitatively different from the tokens used to construct the source code of the program.” – (Biggerstaff et al. 1993, sec. 1)

In substance, this means to explain – and thereby to understand – at an abstraction level higher than the syntactical one what a computer program does as well as how. As programming involves mappings from an application domain, understanding a completed program involves reconstructing these mappings (Brooks 1983). This could be accomplished by a programmer through cognitive processes, which can be defined along the concepts of a mental model and a cognitive model:

“A mental model describes a developer’s mental representation of the program to be understood whereas a cognitive model describes the cognitive processes and temporary information structures in the programmer’s head that are used to form the mental model.” – (Storey 2005, sec. 2.1)

“[...] although programmers might use different comprehension strategies in order to understand a program, they must construct a number of mental models at different levels of abstraction [...]” – (Sim 2010)

These cognitive processes can follow a top-down, a bottom-up or a mixed comprehension strategy (Storey 2005). In the case of the top-down strategy, an initial hypothesis about the aspects of the computer program according to the quotation of Biggerstaff et al. above is generated. By means of subsidiary hypotheses a hypothesis is refined stepwise. Thus, the hypotheses
can be verified and revised recursively. (Brooks 1983) In the case of the bottom-up strategy, source code statements are grouped to so-called chunks, which are subsequently grouped to larger chunks consisting of chunks, and so forth. While the initial chunks are the smallest units a high-level interpretation can be associated with, the larger chunks strive for a high-level understanding of the entire computer program. (Shneiderman & Mayer 1979) To perform a top-down or a bottom-up comprehension strategy, there are two different approaches to browse through the source code – depth-first and breadth-first (Storey 2005) – which are self-explanatory.

2.2 Agile software development

Agile software development describes a category of software development methodologies that share some “agile” properties. These have been phrased by agile values and agile principles in the Agile Manifesto (Beck et al. 2001a), which will be reviewed according to the aim of this thesis in sections 2.2.1 and 2.2.2. Due to the breadth of agile software development in practice, caused by the different methodologies, it is hard to give an introductory description of agile software development that is concise as well as universally valid. Most suitable in the context of this thesis might be the perception of agile software development as an empirical approach opposed to a traditional plan-driven approach (Highsmith & Cockburn 2001). The latter is also known as a heavyweight approach as that “assumes that problems are fully specifiable, and that an optimal and predictable solution exists for every problem” (Nerur et al. 2005, p. 74). In contrast to that, the former – a lightweight approach – bases on “short inspect-and-adapt cycles and frequent, short feedback loops” (Williams & Cockburn 2003, p. 40). This kind of iterative and incremental software development is carried out by self-organized teams with a strong collaboration with the customer (Glazer et al. 2008).

Further characterizing for agile software development in general is Extreme Programming (XP). Even though being a specific agile software development methodology, it initially was the most widely used one (Dybå & Dingsøyr 2008). Therefore, XP and its practices, which will be outlined in section 2.2.3, undoubtedly had a considerable impact on the agile practices, which were, compared with the agile values and principles, only loosely specified. They will be subsequently outlined in section 2.2.4. Additionally to XP, Scrum will be outlined as a further agile software development methodology (section 2.2.5) because the empirical research for the thesis was conducted in a software organization that has applied Scrum (see section 5).

Besides these practice-oriented characterizations of agile software development, there exist also theoretically more substantiated characterizations. Worth mentioning among those are Adaptive Software Development (Highsmith 1999) and Lean Software Development (Poppendieck & Poppendieck 2003). However, these are analogous categorizations of software development methodologies themselves and they are only partly congruent with agile software development. Therefore, they were not considered for this thesis. They would rather have broaden the topic agile software development even more.

2.2.1 Agile values

The agile values formed the Agile Manifesto (Beck et al. 2001a). These were:
1. Individuals and interactions over processes and tools
2. Working software over comprehensive documentation
3. Customer collaboration over contract negotiation
4. Responding to change over following a plan

In practice, there is often a misinterpretation of the agile values. Glazer et al. (2008) stated about that:

“The Agile Manifesto is frequently read in such a way [...] where the things on the right (items commonly found in too many plan-driven [...] environments) are not merely valued less, they effectively have zero value.” – (Glazer et al. 2008, p. 16)

“The Agile Manifesto has been frequently cited [...] as justification for not having processes, for not documenting [...] and for not having plans.” – (Glazer et al. 2008, p. 16)

Certainly, this was not intended, as the manifesto literally stated:

“That is, while there is value in the items on the right, we value the items on the left more.” – (Beck et al. 2001a)

This common misinterpretation becomes crucial when considering the apparent contradiction between agile software development and documentation, which underlies this thesis. Especially the second agile value is problematic. Hence, the most important and succinct reflections by Cockburn (2002) and Martin (2003) on this value are quoted here, while their reflections on the other values are omitted:

“The working system is the only thing that tells you what the team has built.” – (Cockburn 2002, p. 217)

“The code does not lie about what it does. It may be hard to extract rationale and intent from the code, but the code is the only unambiguous source of information.” – (Martin 2003, p. 5)

“Software without documentation is a disaster. [...] However, too much documentation is worse than too little.” – (Martin 2003, p. 5)

“Documents can be very useful, [...] but they should be used along with the words just enough and barely sufficient.” – (Cockburn 2002, p. 217)

The apparent contradiction between agile software development and the second agile value is not deepened further here. This will be done later (see section 3 ff.) because here, only the contradiction in principle is necessary for providing a first overview of the agile values in context of the thesis.

2.2.2 Agile principles

For concretizing the agile values, the Agile Manifesto was supplemented with twelve agile principles:
1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer’s competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity – the art of maximizing the amount of work not done – is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

(Beck et al. 2001b)

Likewise to the outline of agile values (see section 2.2.1), reflections by Cockburn (2002) and Martin (2003) on principles that are obviously related to the apparent contradiction between agile software development and documentation are quoted here, while their relation to the contradiction in detail will be investigated later. The principles that definitely are obviously related to the contradiction are the sixth and the seventh.

“The primary mode of communication is conversation.” – (Martin 2003, p. 7)

“Documents may be created, […] but they are not the default.” – (Martin 2003, p. 7)

“Rely on the honesty that comes with running code rather than on promissory notes in the form of plans and documents.” – (Cockburn 2002, p. 220)

“Agile projects […] don’t measure their progress […] by the volume of documentation that has been produced or by the amount of infrastructure code they have created.” – (Martin 2003, p. 7)

2.2.3 Extreme Programming (XP)

Extreme Programming (XP) was mainly coined by the book of Beck (1999). However, as an introductory description, the summary of XP by Highsmith (2002) is quoted here:

“XP preaches the values of community, simplicity, feedback, and courage. Important aspects of XP are its contribution to altering the view of the cost of change
and its emphasis on technical excellence through refactoring and test-first development. XP provides a system of dynamic practices, whose integrity as a holistic unit has been proven. XP has clearly garnered the most interest of any of the Agile approaches.” – (Highsmith 2002, p. xxxiii)

Analogously to agile values and agile principles, there are XP values and XP principles. The latter have been loosely related to the former (Beck & Andres 2004). Anyway, sufficient to grasp the fundamentals of XP are the mentioned XP practices, with which the XP values and principles can be implemented. These XP practices originally were:

1. The Planning Game
2. Small releases
3. Metaphor
4. Simple design
5. Testing
6. Refactoring
7. Pair programming
8. Collective ownership
9. Continuous integration
10. 40 hour week
11. On-site customer
12. Coding standards

(Beck 1999, pp. 47-53)²

At this point only those XP practices that furthermore form the most common agile practices (see section 2.2.4) are concisely explained. Those are the first, the fifth, the sixth, and the seventh XP practice:

**The Planning Game** In XP, you plan the software development incrementally (Beck 1999, p. 9). One planning strategy is that “you could start development with a simple plan, and continually refine it as you went along” (Beck 1999, p. 54). Another one is prioritizing user stories (Beck 1999, p. 47). User stories are descriptions of valuable functions a program should have from the perspective of the user, written in natural language (Cohn 2004, ch. 1). An example is illustrated in figure 1.

**Testing** There are employed “unit tests written by the programmers to convince themselves that their programs work the way they think the programs work” as well as “functional tests written by (or at least specified by) the customers to convince themselves that the system as a whole works the way they think the system as a whole should work” (Beck 1999, p. 45). This is done preferably with a Test-driven Development approach (TDD), i.e. writing such tests, which are automated, before writing the code of the program (Beck & Andres 2004, p. 31). The unit tests test individual units of the source code, which are of smallest possible size. In the case of OOP, these units are usually the methods of a class. (Weyuker 2002) Listing 1 shows an exemplary unit test, written with the JUnit framework (ObjectMentor 2010). By executing that unit test, a unit is tested in three

---

²In the second edition of the book (Beck & Andres 2004), the XP practices were grouped and renamed. But the original twelve practices from the first edition of the book (Beck 1999) remain the most often quoted variant.
A company can pay for a job posting with a credit card.

Note: Accept Visa, MasterCard, and American Express. Consider Discover.

Figure 1: Example of a user story

taken from Cohn (2004, p. 19)

different ways – represented by the three visible test methods. If any of the assertions about the expected result – made through the calls named assert...() – prove to be false, a failure is reported. Because for writing unit tests the source code needs to be accessible and because generally the software developers with knowledge about the code write unit tests, this kind of testing is known as white-box testing (Weyuker 2002). In contrast to that, functional tests constitute black-box testing. For that kind of testing particularly the mentioned user stories are (re-)used. In this context, they are also called acceptance tests (Cohn 2004, ch. 1). Nonetheless, functional tests can be incorporated into unit tests.3

Refactoring The source code is restructured, but without changing its behavior. The purpose is, for example, to remove duplication or to add flexibility. To ensure the same behavior after or during the refactoring the tests can be executed. This additional effort facilitates maintainability and extensibility of a program. (Beck 1999) As an example for refactoring the code from listing 1 could serve, because that was actually an example for the refactoring of the code of a unit test by Martin (2009), who stressed that the code of unit tests should be refactored likewise with the code to test. The original – unrefactored – version of the unit test, shown in appendix A, is characterized mainly by redundancy and an obfuscating structure.

Pair Programming All code is written by two programmers at one computer. Each of them has a different role: One focuses on implementing a part of the program by typing, while the other one thinks likewise about how to implement, but rather without actually doing that. Therefore, one can reflect on a particular programming task more detailed (Beck 1999) However, this is much more than only passive watching, it is “a dialog between two people trying to simultaneously program (and analyze and design and test) and understand together how to program better” (Beck 1999, p. 81). According to Beck, it is not supposed as a tutoring session hold by an experienced programmer for a less experienced programmer, but this might be a beneficial side effect, which can be enhanced further by switching and re-arranging pairs. Certainly, this XP practice encompasses a high interpersonal conflict potential. For example, it is not unlikely that two people in a pair own a considerably different extent of critical thinking.

3For example, this is the case in the listing in appendix A, where the FitNesseContext objects indicate the use of the acceptance testing tool FitNesse (Martin et al. 2010).
Listing 1: A unit test written with the JUnit framework (excerpt of a Java class)\textsuperscript{a}

```java
public void testGetPageHierarchyAsXml() throws Exception {
    makePages("PageOne", "PageOne.ChildOne", "PageTwo");
    submitRequest("root", "type:pages");
    assertResponseIsXML();
    assertResponseContains("<name>PageOne</name>", "<name>PageTwo</name>",
        "<name>ChildOne</name>);
}

public void testSymbolicLinksAreNotInXmlPageHierarchy() throws Exception {
    WikiPage page = makePage("PageOne");
    makePages("PageOne.ChildOne", "PageTwo");
    addLinkTo(page, "PageTwo", "SymPage");
    submitRequest("root", "type:pages");
    assertResponseIsXML();
    assertResponseContains("<name>PageOne</name>", "<name>PageTwo</name>",
        "<name>ChildOne</name>");
    assertResponseDoesNotContain("SymPage");
}

public void testDataGetAsXml() throws Exception {
    makePageWithContent("TestPageOne", "test page");
    submitRequest("TestPageOne", "type:data");
    assertResponseIsXML();
    assertResponseContains("test page", "<Test");
}

\textsuperscript{a}taken from Martin (2009, pp. 126-127)
```
2.2.4 Agile practices

While agile values (see section 2.2.1) and agile principles (see section 2.2.2) are well-defined, agile practices are not. Even though this term is used often in literature, there virtually does not exist any distinct definition of it. Nonetheless, agile practices are commonly considered as concrete activities and techniques that represent the implementation of the agile principles. (Sidky et al. 2007) According to Sidky et al., agile practices are not rarely viewed as more or less identical with XP practices. There is at least an agreement that the most popular XP practices form some of the agile practices. Those are, as mentioned before (see 2.2.3), testing, refactoring, pair programming, and the planning game – detached from XP better known as adaptive planning. That becomes evident when reading literature about agile software development: Those four agile practices are frequently cited. For example, Martin (2003) even devoted a separate chapter to each of these practices in his book. Moreover, the technique of user stories introduced by XP had a remarkable impact on agile software development in general. Consequently, it is often regarded as an independent agile practice (Cockburn 2002).

2.2.5 Scrum

Scrum is a project management framework and has been used to successfully deliver a wide range of complex products (Highsmith 2002, p. xxxii). However, it has been found mainly in software development. And as its application enforces a time-boxed and team-oriented development, it counts as an agile methodology. (Schwaber & Sutherland 2010) Essential for facilitating the time-boxed development is the so-called Sprint:

“A sprint [...] is an iteration of one month or less that is of consistent length throughout a development effort. All Sprints use the same Scrum framework, and all Sprints deliver an increment of the final product that is potentially releasable. One Sprint starts immediately after the other.” – (Schwaber & Sutherland 2010, pp. 4-5)

Involved in a Sprint are all members of a Scrum Team, who fulfill one or more of the following roles:

**Scrum Master** Ensures that the development process is carried out with adherence to the Scrum framework. Usually, the Scrum Master is well-experienced with Scrum in practice as teaching the Scrum Team and optimizing the organizational environment are crucial ambitions.

**Product Owner** Ensures the value of the work done by the Scrum team and the visibility of the development progress to its members as well as to the organization. This is mainly realized by means of the Product Backlog (described below). The Product Owner might be, for example, a product manager in a software company but to avoid conflicts of interests, never the Scrum Master at the same time.

**Team** Is a group preferably of about seven people who work cross-functionally to develop a product in increments. Each member of this group has a particular skill, such as programming, quality control, business analysis, architecture, user interface design, or data
base design. Furthermore, team skills are required because the group is intended to be self-organized.

(Schwaber & Sutherland 2010)

Other roles are not designated in Scrum. Hence, stakeholders like customers or managers of the organization are not part of the Scrum Team, even though a Scrum Team can consult them in meetings (Schwaber & Sutherland 2010). Several such meetings are predetermined, most important are the Sprint Planning meeting and the Daily Scrum:

“The Sprint Planning meeting is when the iteration is planned. It is time-boxed to eight hours for a one month Sprint. [...] The Sprint Planning Meeting consists of two parts. The first part is when what will be done in the Sprint is decided upon. The second part (a four-hour time-box for a monthly Sprint) is when the Team figures out how it is going to build this functionality into a product increment during the Sprint.” – (Schwaber & Sutherland 2010, pp. 11-12)

“Each Team meets daily for a 15-minute inspect and adapt meeting called the Daily Scrum. The Daily Scrum is at the same time and same place throughout the Sprints. During the meeting, each Team member explains: What he or she has accomplished since the last meeting; what he or she is going to do before the next meeting; and what obstacles are in his or her way.” – (Schwaber & Sutherland 2010, p. 15)

Besides the Sprint Planning meeting and the Daily Scrum, there are further meetings: the Sprint Review meeting, the Sprint Retrospective meeting, and the Release Planning meeting (Schwaber & Sutherland 2010). But they are not crucial for grasping the fundamentals of Scrum. Thus, they are not explained here.

The remaining work to be done at a certain point in time for completing a product is expressed by Backlogs, which exist in three different variants:

**Sprint Backlog** Consists of the tasks that are intended to be done in the next Sprint. These tasks are determined by the team in the Sprint Planning meeting. Accomplishing them serves as a goal for the Sprint – to complete parts of a product, which are described in a **Product Backlog**.

**Product Backlog** Consists of the work related to the development of a product, such as product features, functionality, infrastructure, architecture, or technology work. Compared with the Sprint Backlog, it is more loosely stated. The Product Owner is responsible for maintaining it throughout the development process of the product.

**Release Backlog** Is a subset of the Product Backlog that is planned for a release. The Product Owner may adjust the corresponding Product Backlog according to the progress achieved in the last Sprint.

(Schwaber & Beedle 2002)
3 Research framework

For carrying out research for this thesis, it was seen as essential to further specify the requirements and delimitations in respect of the research goals and the supposed results. For that, requisites are presented in this section, which had to be met in the subsequent research – the literature review (see section 4) as well as the survey studies (see section 5). These requisites are deduced from the research goals (section 1.3), and the implications of documenting for program comprehension and practicing agile software development as introduced in section 2.2. In the case of agile software development, only non-trivial deductions – particularly from the second agile value – are incorporated in a requisite. Generally, solely issues are addressed that have not been handled exhaustively so far and that result in non-obvious, relevant findings. Certainly, a rationale for each requisite and, where appropriate, for the hypotheses made for that is given. In this context, it was paid particular attention to the referenced literature used for reasoning: Using the same literature for substantiating the requisites and for extracting potential techniques or tools for the literature review would lead to a tautology because the latter has to meet the former. The order in which the requisites are listed here – each in a separate subsection – conforms to an understandable advancement from one requisite to another. For highlighting them within the text, they will be written in “CamelCase” notation with small capitals throughout the thesis.

3.1 The LowLevelContext requisite

Obviously, all documentation techniques and tools that are considered for this thesis’ results have to be attached to the software in development on a low-level context, i.e. the source code of that. This can be deduced directly from its purpose of supporting program comprehension, which is largely a matter related to programming. Software developers, those who perform that, generally consider it as crucial that documentation directly relates to the source code (Forward & Lethbridge 2002b). Furthermore, this can also be deduced from the second agile value: As elaborated in section 2.2.1, the source code is crucial for documentation in general. This was already expressed by “the code is the documentation” in section 1.2. Therefore, it might appear like a trivial deduction, but as that is of considerably high importance, it is emphasized here in the form of a requisite. Besides, a low-level context in agile software development is not only self-evident due to the interpretation of source code as a kind of documentation. Actually, agile software development itself “lives” in a comparable low-level context: As elaborated in section 2.2.4, most agile practices are directly concerned with programming and not with software development at large.

3.2 The HighLevelBenefit requisite

As mentioned in the introduction, a documentation attached to a low-level context of software development could and should also provide a system-level understanding for involved stakeholders, who are not (necessarily) software developers. These could be, for example, product managers in a software organization or the customer for whom the software is developed. From now on, those stakeholders are called project stakeholders to distinguish them from developers
as suggested by Aguanno (2005). Characteristic of the notion of a system level that is used here is the meaning of a documentation related to a context at a higher level, which is understandable for project stakeholders. The purpose behind such a documentation is to add a benefit within its high-level context. Such a benefit could be:

1. An understanding that makes the progress of software development more visible. Regarding the software in development as a product, that would be beneficial for the project or product management. For example, this could possibly result in a faster completion of the software product.

2. An eventually improved quality of the developed software product. Such a quality improvement could also eventuate through an additional value to the software product. As mentioned in the introduction, a prime example for that is a reference for an API, with the API itself as the software product.

A high-level benefit of either the first or second type could be induced through a documentation artifact extracted from the documentation that was originally supposed to support program comprehension. This could be done by transforming that low-level documentation to a high-level documentation or by generating another high-level documentation artifact from it. Conceivable is also that – in the best case – low-level and high-level documentation are the same documentation artifacts.

Contrary to the LOWLEVELCONTEXT requisite above, this requisite is not self-evidently obligatory according to the second agile value. Besides, it is not as clearly deducible from that. Nonetheless, the requisite can be deduced from that agile value in the case of the first type of a high-level benefit by considering the agile principles from section 2.2.2. Crucial is the seventh agile principle (“working software is the primary measure of progress”). This principle applies also to the documentation because that was determined to be rooted in the source code through the LOWLEVELCONTEXT requisite. Therefore, it is implied to deliver apart from the working source code also the documentation in its transformed or generated variant to the customer.

In the case of the second type of a high-level benefit, it can be argued that, certainly, a documentation supporting program comprehension itself probably leads to an eventually higher quality of the developed software (Maletic & Kagdi 2008). This can be seen as a consequential benefit of a potentially improved program comprehension. Nonetheless, a thus deduced rationale is not conform to agile values or principles in particular. Hence, a rationale deduced from agile principles like that in the case of the first type of a high-level benefit is suggesting. In this respect, a difficulty occurs: While the first type deals with the properties of the software product during its development, the second type deals with its properties after its completion. The former is principally covered directly by agile principles, as those more or less talk about processes (Martin 2003), whereas the latter is rather covered indirectly by them. However, a compliance with the ninth agile principle (“Continuous attention to technical excellence and good design enhances agility.”) leads implicitly to a higher quality of the final software product because, according to the reflection on this agile principle by (Martin 2003, p. 8), “all agile teams members are committed to producing only the highest quality code they can.” If this

---

4If it leads analogously to the first type of a high-level benefit to a faster development, is unclear. There are findings that argue for that (Sayyad-Shirabad et al. 1997) as well as findings that argue against that (Ratanotayanon et al. 2006, Xiaomin et al. 2004).
high quality is achieved by creating a further documentation artifact, the second agile value is not necessarily infringed by that: It is conceivable that the documentation artifact is extracted from the original documentation for program comprehension in an sufficiently lightweight way conform to agile software development.

Besides an extracted documentation artifact for the purpose of achieving a high quality of the software product, a further purpose of an extraction from the documentation for program comprehension is evident: A high-level benefit induced through a documentation artifact obtained by such an extraction definitely helps a developer to justify the effort of (continually) documenting on a low level. If the project stakeholder discernibly benefits from this documentation, that stakeholder is most likely convinced of that. Thus, the effort of documenting for program comprehension is possibly granted at a higher probability – if communicated appropriately to the customer. In the context of agile software development, this scenario is of particular significance: Expectedly, a documentation for supporting program comprehension is rather not initiated by the customer. This is not problematic as long as the customer appreciates quality to a certain extent. Then the treated ninth agile principle comprises the motivation for a documentation for program comprehension without contradiction to the customer’s requests. But if the customer does not appreciate quality to such an extent, this principle is outweighed by the first principle that puts customer satisfaction in the first place (see section 2.2.2). In this case, the need for a justification, which could change the customer’s mind, is increased. Therefore, a high-level documentation extracted for that purpose from the original documentation on a low-level context is regarded as a prescription.

### 3.3 The NoSeparateArtifact requisite

Generally, documentation can be a separate artifact of a software product apart from a developed computer program that actually constitutes that software. This is the case if the customer or a product manager explicitly requests such a documentation to be developed as part of the software product, e.g. a user manual for the software. Contrary to that, it would be implausible to request a likewise separated documentation for the purpose of supporting program comprehension. This becomes obvious when considering that the possibility to convince a project stakeholder of the documentation effort was given as a rationale for the HIGHLEVELBENEFIT requisite. A stakeholder who is not a developer most likely does not even know what program comprehension is at all. Regardless of that, program comprehension is a process that occurs during the development of the software. Therefore, it can hardly be praised as relevant for the resulting software product itself. Otherwise, it would be inevitable to extend the concept of a documentation for program comprehension to the notion of a re-documentation (Chikofsky & Cross II 1990). Practicing re-documentation as a process largely complies with program comprehension in its essence:

“Usually, programmers acquire knowledge about the system by means of code reading [...] In order to avoid that such knowledge gets lost over time, it must be stored persistently in some format (re-documentation), such as, for example, a set of design diagrams.” – (Torchiano et al. 2010, p. 15)

However, for program comprehension, it is not essential to document the consequently acquired program knowledge. Doing so anyway would infringe the tenth agile principle (see section
2.2.2), which is about that work which should not be done. In this context, particularly the so-called YAGNI principle – originally proposed in connection with XP – makes that clear, as it stands for “You’re not gonna need it!” (Jeffries 1998). A designated use of re-documentation is to document “legacy software” that was once developed without any documentation (de Souza et al. 2007). However, in the case of an iteratively and incrementally developed software such a scenario is rather atypical. There are no principles or practices in agile software development that would be in need of re-documentation. Contrary to that, reverse engineering – a process encompassed more or less by re-documentation (Chikofsky & Cross II 1990) – is not likewise contradictory to agile software development because it does not set specific outcomes that might be unneeded:

“Reverse engineering is the process of analyzing a subject system to identify the system’s components and their interrelationships, and create representations of the system in another form or at a higher level of abstraction.” – (Chikofsky & Cross II 1990, p. 15).

Still, there is the problem that creating those representations – basically documentation – can be carried out automatically only to a certain extent, which can vary considerably depending on the concrete case (Torchiano et al. 2010). It is unclear whether such semi-automatic reverse engineering is justified because there is the ubiquitous danger that the documentation effort outweighs the supposed profit in terms of the second agile value. For meeting the HIGH-LEVEL.BENEFIT requisite, it is suggesting to employ reverse engineering for the required transformation or generation of a documentation artifact. Then, it might be tempting to overdo documentation for program comprehension solely because the reverse-engineered high-level documentation could consequentially be particularly valuable for a project stakeholder. However, the HIGH-LEVEL.BENEFIT requisite does not ask for such a violation because it states nothing about the value of the high-level documentation and whether that could or should outweigh the value of the original purpose of supporting program comprehension. To prevent that, it is evidently necessary to prescribe that a low-level documentation must not be a separate artifact of a software product, even though the extracted high-level documentation could be eligible for being that. The high-level documentation might be explicitly requested by the customer or a representative, but then it should still be regarded as of secondary importance. Reconsidering the given example of a reference for an API, you could say – freely based on the second agile value – for an exemplification of this requisite that a working API is definitely more important than a comprehensive documentation of it.

3.4 The PrimarilyProgramComprehension requisite

Necessarily, any technique or tool to be considered in this thesis has to serve the purpose of supporting program comprehension to a certain degree. Although a vast part of documentation in software development might meet this requirement, those techniques and tools have to serve primarily the purpose of supporting program comprehension. Serving exclusively the purpose of supporting program comprehension obviously is a too strict limitation, because a vast part of feasible documentation for that might serve also other purposes, intentionally or not. Anyway, the former limitation is regarded as a complement to the NOSEPARATEARTIFACT requisite.
Independently from that requisite, this limitation is inevitable due to a practical reason: For proving an effect of the application of a technique or tool by relating it to accepted theories, a starting point for doing research in the field of documentation in software development is helpful – here the purpose of supporting program comprehension. Without a narrowing through that starting point, this might be impracticable as documentation in software development obviously is an enormously broad topic. Certainly, it has to be emphasized that it is only a starting point: Holding that too strictly could lead to overlooking documentation effects that are not primarily but still significantly beneficial for program comprehension.

Alternatively, typical documentation artifacts induced by the application of a specific documentation-grounded software development approach like, for example, Model Driven Development (MDD) could be considered. In the case of MDD, the existence of certain documentation artifacts like, for example, visual models in the form of diagrams can be presumed. These could be examined regarding their usefulness for program comprehension. Nonetheless, this procedure would narrow the broadness of documentation too much as it cannot be assumed that a software organization generally applies a single software development approach. Contrary to that, program comprehension – as a process – is ubiquitous in software development regardless the approach or methodology. Analogous considerations exist in further contexts that depend particularly on the type of the software organization. In the case of an industrial software organization, the branch of business it belongs to is crucial for that. For example, in a branch it could be the industry standard to practice system development beside software development. This would presumably come with documentation itself. Another example is the use of Domain-Specific Languages (DSL), which surely would influence the nature of documentation, at least in a technological manner.

### 3.5 The Arbitrary Agile Methodology requisite

The field of agile software development consists of an open set of various methodologies. So far, most of the known agile software development methodologies have existed before the publishing of the Agile Manifesto (Abrahamsson et al. 2003). That has established a common framework shared by all of those methodologies, each emphasizing some particular agile values and principles over others by implementing them in different ways. Considering each methodology with its particular emphasis would certainly go beyond the scope of this thesis. Moreover, its generality would be compromised, as the set of agile software development methodologies is not predefined and remains open to further methodologies emerging in the future. As given in section 1.3, a technique or tool for documenting for program comprehension has to be suitable for any agile software development methodology. Hence, it has to be examined in respect of the Agile Manifesto, i.e the agile values and principles (see sections 2.2.1 and 2.2.2).

Abrahamsson et al. (2003) exposed the different emphases of the agile software development methodologies along a typical software development life-cycle regarding, among other things, a methodology’s support of project management and its practical guidance for getting carried out. XP and Scrum – the two concrete methodologies presented in sections 2.2.3 and 2.2.5 –

---

5 An application of MDD coupled with an agile software development methodology – known as Agile Modeling – is principally feasible. This will be addressed in section 4.2.
revealed as quite different, although both address the same six life-cycle stages\(^6\) from totally nine stages\(^7\). While XP covers these stages completely with its concrete XP practices, i.e. practical guidance, it does not relate them to project management at all. In contrast to that, Scrum is characterized by a distinct focus on project management throughout the stages while giving almost no practical guidance.

Agile software development methodologies could also be distinguished by their conceptual properties like, for example, their \textit{scalability} or their \textit{tolerance to variations} (Cockburn 2002, ch. 4). Considering scalability, XP is recommended to be applied in teams of two to ten people (Beck 1999), while Scrum is much more scalable through \textit{Scrum-of-Scrum}, where each Scrum team sends a team member to a superordinate Scrum team (Cohn 2007). Furthermore, a particular methodology itself can differ from its “standard definition” in its application, what is meant with tolerance to variations. For example, XP is well-known for being applied partly – in the form of a \textit{tailored XP} – because in some cases it is impractical to apply \textit{all} XP practices in their given definition (Hussain et al. 2008). A methodology can also be applied with influence from another methodology or in combination with that. In the case of Scrum, these could be, for example, Scrum combined with XP (Kniberg 2007) or Scrum combined with \textit{Kanban}\(^8\), which is also known as \textit{Scrumban} (Ladas 2009, Kniberg 2010). Nonetheless, in respect of generality it would be unreasonable to consider for this thesis an application of an agile software development methodology that differs from its “standard definition” or that is combined with another methodology.

\section*{3.6 The ArbitraryProgrammingLanguage requisite}

Regarding the desired generality of this thesis’ results, conditions for the programming language employed in the software development have to be determined likewise to the \texttt{ARBITRARYAGILEMETHODOLOGY} requisite. This can be deduced from the relation of program comprehension to programming. Certainly, a technique or tool for documenting for program comprehension should be independent from a specific programming language or a programming platform that embraces several programming languages.\(^9\)

However, the \textit{programming paradigm} that comes into question must be fix, because considering \textit{all} known paradigms would go beyond the scope of this thesis. The process of program comprehension is considerably unique for each paradigm. For example, object-oriented programming (OOP) eases that in many respects compared to procedural programming – despite not turning program comprehension expendable (Burkhardt et al. 1997). Therefore, OOP is determined as the concrete paradigm that comes into question. OOP is not only the most commonly applied programming paradigm today, also the research in agile software development and the research in documentation consider exclusively this paradigm for the most part.\(^10\)

\footnote{\textit{requirements specification}, \textit{design}, \textit{code}, \textit{unit test}, \textit{integration test}, and \textit{system test}}

\footnote{additionally \textit{project inception}, \textit{acceptance test}, and \textit{system in use}}

\footnote{\textit{a concept related to just-in-time production} (Lu 1989), thus, as a software development methodology more related to \textit{Lean Software Development} (Poppendieck & Poppendieck 2003) than to agile software development}

\footnote{\textit{For example, the Java Virtual Machine} is devoted to \textit{Java} as its “native language”, but also to \textit{Scala} and other programming languages (EPFL 2010).}

\footnote{In this context, it should be mentioned that, in contrast to those topics, research in program comprehension has been done mainly in respect of procedural programming and not OOP (Burkhardt et al. 1997).}
Apart from the source code of the computer program, which essentially constitutes the developed software product, there are often produced additional source code artifacts like, for example, shell scripts for the deployment of the program, or other code-like artifacts such as descriptively composed configuration files (Spinellis 2010). However, those artifacts should not be considered because there is no general agreement which types of artifacts are commonly included in a software product.
4 Literature review

The literature review for this thesis was carried out according to the recommendations for conducting a literature review in information systems and computing disciplines by Oates (2006, ch. 6). It is elaborated in this section the search method for finding relevant literature (section 4.1), followed by the results of an inspection of the found literature for obtaining the significant part of that (section 4.2). The thus obtained literature is summarized and discussed in detail if it leads to potential research results for the thesis, i.e. techniques or tools for documenting for supporting program comprehension in agile software development (sections 4.3 to 4.8). The discussion of each technique or tool is done systematically: First, its relation to program comprehension is addressed, then its relation to agile software development. As part of that as well as subsequently, it is deduced whether the requisites from the research framework (see section 3) are met and, when indicated, under which constraints. Finally, the relevance of the technique or tool for the thesis is remarked. Additionally to the textual consideration of the requisites in this section, an overview of the findings can be found in table form in appendix C.

4.1 Search method

The topic of this thesis, determined by its title, was split into separate research concepts, which obviously are constituted by these three subjects: Program comprehension, documentation, agile software development. To illustrate them here, it is assumed that they can be embraced by sets in a mathematical sense. Thus, they can be depicted as a Venn diagram like in figure 2. The sets in the diagram contain corresponding literature, i.e. research works, and are named with acronyms in the following way: P for program comprehension, D for documentation, and A for agile software development. Their intersections are named with a corresponding combination of those in alphabetical order, e.g. AP for the intersection of A and P. However, the relative proportions of the sets or their intersections to each other in the diagram do not correspond to actual quantities. For example, A could feasibly more extensive than P in terms of all available research works of these research concepts, albeit the equal proportions of the sets in the diagram. Or, ADP could be – and presumably is – much more narrowed than PD, even though the intersection ADP is relatively larger in the diagram.

The diagram’s purpose is to depict the relationships between the research concepts and the feasible options to combine them according to the supposed results of this thesis. Certainly, this lies within the intersection ADP, more precisely in the left-uppermost edge of that, as the diagram embodies a further meaning depicted by the two arrows: The sets as well as the elements they contain are orientated towards the nature of processes – more heavyweight or more lightweight – and the nature of the artifacts created by these processes – more relevant to software developers or more relevant to customers. Any applied agile software development methodology should be kept as lightweight and as customer-oriented as possible, as it is directly deducible from the agile values, particularly from the first and the third one (see section 2.2.1). Hence, striving left-uppermost within ADP is binding.

For establishing this intersection, it was suggesting to narrow down the research concepts step-by-step, i.e. a research concept was narrowed down by the next research concept in the sequence. For that, six different approaches existed, each determined by a sequence of research
Figure 2: Venn diagram with research concepts

Table 1: Narrowing down sequence for research concepts

<table>
<thead>
<tr>
<th></th>
<th>1st</th>
<th>2nd</th>
<th>3rd</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>D</td>
<td>P</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>P</td>
<td>D</td>
</tr>
<tr>
<td>3</td>
<td>D</td>
<td>A</td>
<td>P</td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td>P</td>
<td>A</td>
</tr>
<tr>
<td>5</td>
<td>P</td>
<td>A</td>
<td>D</td>
</tr>
<tr>
<td>6</td>
<td>P</td>
<td>D</td>
<td>A</td>
</tr>
</tbody>
</table>
concepts to examine – permutations of A, D and P (see table 1). All six approaches were carried out by a joint set of keyword searches. For that, the Google Scholar search engine (Google 2010b) and several research databases were used:

1. IEEE Xplore (IEEE 2010)
2. ACM Digital Library (ACM 2010)
3. ScienceDirect (Elsevier 2010)
4. SpringerLink (Springer 2010)
5. Wiley Online Library (Wiley-Blackwell 2010)

The keywords used for each search singly as well as combined together were:

1. “Agile Software Development”
2. “Agile Documentation”
3. “Agile Documenting”
4. “Documentation”
5. “Program Comprehension”
6. “Program Understanding” (as synonym of “Program Comprehension”)

Subsequently, a search with the latter three keywords was repeated by prefixing the term “agile” to strengthen the reference to agile software development. Likewise, this term was used for searches with keywords that were contained multiple times in the obtained research works and that apparently refer to research fields related to the represented three ones. The purpose behind such a search was not to obtain further research works but to confirm the boundaries to other distinct research fields. These keywords were:

1. “Reverse Engineering”
2. “Re-Documentation”
3. “Knowledge Management”
4. “Knowledge Sharing”
5. “Modeling”

As a large part of the obtained research works were published in conference proceedings, the most frequently emerged conferences were searched. All articles in the available proceedings of each of these conferences were checked for relevance by reading the title and, optionally, the abstract. These conferences were:

1. Agile Development Conference (AGILE 2010)
2. ACM International Conference on Design of Communication (SIGDOC 2010)
3. IEEE International Conference on Program Comprehension (ICPC 2010)
4. IEEE International Conference on Software Maintenance (ICSM 2010)
5. Working Conference on Reverse Engineering (WCRE 2010)

Likewise, all available publications of certain authors were checked for relevance. These authors were researchers who repeatedly published research works concerning a single subject over a longer period or who were cited multiple times by the other obtained research works. Eventually, to get insights from practice, a book search with the above-mentioned keywords
was conducted via Google Books (Google 2010a). Further, practice-oriented books about agile software development in general from authors (Cockburn 2002, Highsmith 2002, Martin 2003) who were writers of the Agile Manifesto (Beck et al. 2001a) were examined as part of this literature review. Despite deficient scientific grounding, these books are widely accepted by practitioners, as agile software development originally was introduced and coined by those people. And because most research works with sufficient scientific grounding revealed quoting the Agile Manifesto, it was suggesting to consider these books – later publications by the writers of that – as a base for a broader literature review about agile software development.

### 4.2 Inspection results

The found literature was inspected from the perspective of each of the six possible narrowing down sequences from table 1 to obtain research works that comply with that sequence. The results of this inspection are presented here, in the same order as in the table.

**A,D,P** While A itself is an extensive research concept, the ARBITRARYAGILEMETHODOLOGY requisite limited A considerably. Only the common properties of all agile software development methodologies could be considered for narrowing down. Certainly, these have been constituted by the agile values (see section 2.2.1), principles (2.2.2), and – to a certain extent (due to their vagueness) – practices (2.2.4). Consequently, the narrowing down by D for attaining the intersection AD from A had to be rooted in agile values and principles. The literature review revealed that there were almost no research works about documentation available which embody this notion, presumably due to the apparent “dis-like” of documentation expressed by the second agile value. Salient in this context was the online publication by Ambler (2010a), who presented agile documentation principles and practices that were directly deduced from agile principles and practices. However, his work was strongly influenced by Agile Modeling (Ambler 2010b), which has been his major interest.11 Hence, some parts of his work are not usable for this thesis, otherwise the ARBITRARYPROGRAMMINGLANGUAGE requisite would be violated. Even more critical is the fact that his work is of very little scientific grounding. Therefore, it can only be used quantitatively for providing additional confirmations of the documentation concepts of the techniques and tools, besides necessary indisputable confirmations of those through scientifically grounded literature. The latter thus constitutes a qualitative usage. Besides that, the work of Ambler could be used also as an underpinning for the developed research framework in section 3. However, more clean from a scientific standpoint is the focus on the commonly accepted agile principles and practices instead of his agile documentation principles and practices for that purpose. Otherwise, the work of Ambler (2010a) could be used mainly for underpinning the HIGHLEVELBENEFIT requisite. Most notably in this context is his statement that there is a need to “write documentation for people that won’t have access to the source code, or at least don’t have the ability to understand it” (Ambler 2010a, sec. 6.7).

**A,P,D** In opposite to the first narrowing down sequence, there has not been a likewise dislike of P by A. However, the agile values and principles revealed as being too general to

---

11The online publication about agile documentation (Ambler 2010a) was derived from a chapter of the same name from his book about Agile Modeling (Ambler 2010b). The online publication in place of the book chapter is used as a reference throughout this thesis because it was more extensive and up-to-date.
consider them for the subsequent narrowing down by P. Instead, a research work (van Deursen 2001) was found that related program comprehension to the practices from the agile software development methodology XP (see section 2.2.3). Even though the research work itself violates the ARBITRARYAGILEMETHODOLOGY requisite, some findings can be extracted because some of the XP practices are also well-known as common agile practices independently from a specific agile software development methodology (Martin 2003). These as well as their potential relevance for documentation will be included in the forthcoming discussion of the other found promising research works. As mentioned in the introduction, the testing practice embodied by unit tests will be among them (sections 4.5 and 4.6).

D,A,P D itself is not limited by any requisite like A is. Hence, this narrowing down sequence was mainly conducted by reviewing the obtained literature from searches with the keywords “agile documentation” and “agile documenting”. Similarly to Ambler (2010a) in the first narrowing down sequence, the book of Rüping (2003) revealed as being salient. He proposed a relatively concrete and coherent set of solutions for documenting in agile software development in the form of patterns. However, in opposite to Ambler, these patterns were not explicitly related to the agile values or principles as they were aggregated solely from the author’s experience. Furthermore, like the work of Ambler, the work of Rüping is of little scientific grounding. Likewise, it can only be used quantitatively. The patterns referred to for that can be found in appendix D. Again, there emerges the opportunity to use this work for underpinning the research framework. For example, the Code-Comment Proximity pattern (see appendix D) complies largely with the LOWLEVELCONTEXT requisite. Certainly, there were obtained also research works with sufficient scientific grounding. Nonetheless, a large part of these research works can at most be considered as scattered inspirations due to the challenge to meet the ARBITRARYAGILEMETHODOLOGY and ARBITRARYPROGRAMMINGLANGUAGE requisite as well as becoming concrete to some extent for leading to a technique or tool. For example, Selic (2009) did not violate the requisite but only uttered vague ideas, whereas Baptista (2008) proposed an approach that was worked out well, but that also based inevitably on Scrum. However, one promising and concrete proposal was found that meets the requisite (Sauer 2003). It will be delineated and discussed in section 4.8. Because it was not related to program comprehension by the author, this will be done as part of its discussion.

D,P,A In opposite to the narrowing down sequence above, the first two research concepts of this sequence could not be examined conjointly through a single keyword search. While “documentation” and “agile” are terms on the same level, interpretable as opponents, like “heavyweight” and “lightweight”, “documentation” and “program comprehension” are rather companions, like “mean” and “purpose” (see section 1.1). Thus, the results of searches with a combination of these keywords were not worthwhile in terms of an intended intersection DP. Alternatively, a comprehensive examination of D followed by an examination of P would have been suggesting. This was attempted, despite contradicting the motivation for the PRIMARILYPROGRAMCOMPREHENSION requisite (see section 3.4). However, D revealed as being a too broad research concept to narrow it comprehensively without a limitation beforehand like in the first and second narrowing down sequence. Nonetheless, the research works about D as a subject in general, which have been discovered so far (Forward & Lethbridge 2002a,b), can be used for confirming the fundamental concepts of some techniques or tools.
In comparison to A and D, P is a sufficiently narrowed research concept to consider it atomically before any narrowing down by another research concept. While there were clearly recognizable relations from A to P (see the second narrowing down sequence), there was not found anything alike for the opposite direction – from P to A. Apparently, P itself is ubiquitous regardless which particular software development methodology is applied. Therefore, it might not be surprising that only one research work was found that relates P as a concept in general to A as a particular software development methodology (Maletic & Kagdi 2008). Keyword searches which narrowed the already relatively narrowed P further in the direction of A uncovered a research work that presented a highly innovative tool (Ratanotayanon et al. 2009). Actually, this tool was the only found tool that has been explicitly intended to support program comprehension as well as to be applied in agile software development. Ratanotayanon et al. (2009) themselves appraised their tool similarly. Besides that tool, another tool (Buckner et al. 2005) which was intended to support program comprehension and which was considered to be applied in agile software development was found by – remarkably – each of those keyword searches. However, the latter was not addressed likewise explicitly. And eventually, it could hardly be seen as being related to documentation in any way. For this reason, it was ignored, contrary to the tool of Ratanotayanon et al. (2009). That employs user stories, which are common in agile software development (see section 2.2.3), for the purpose of supporting program comprehension. Furthermore, it also embodies a kind of documentation, whereby it lies already in ADP without any subsequent examination of D. In section 4.7 it will be elaborated and discussed.

Like the narrowing down sequence before, this sequence was carried out by examining P atomically. Particularly, techniques and tools for the purpose of P were considered. Subsequently, these were examined whether belonging also to D. Appropriate keyword searches have not led to research works that proposed further techniques or tools for a such an examination. This might be evident, as D proved to be the most common mean for P. Most influential within the intersection PD, as revealed by the many citations of it, is the Literate Programming approach by Knuth (1984). Its elaboration and discussion including a matching to A will be found in section 4.3. It will form the basis for elaborating and discussing the later approach doc comments in section 4.4.

4.3 Literate Programming

4.3.1 Summary

Knuth (1984) opposed his idea of Literate Programming to the then praised Structured Programming (Dahl et al. 1972). He proposed to further improve the comprehensibility of a computer program by interweaving its source code with a textual explanation of the program logic. The concept behind this combination of source code and text was to consider a program as a “work of literature” when writing it:

“Instead of imagining that our main task is to construct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.” – (Knuth 1984, p. 97)
In fact, Knuth discovered a way to explain chunks of a source code (see section 2.1) on a more human-comprehensible abstraction level through a sole text. This text was written in natural language and had the form of logically structured prose. By means of that, it was possible to arrange the chunks on a syntactical level along their explanation. Knuth entitled such interwoven source code and text as literate programs. He introduced a computer language (Knuth 1983) that could be used to create them. It comprised the Pascal programming language for enabling the implementation of the actual program logic. For enabling the implementation of the textual explanation, the TeX typesetting environment (Knuth 1979) was comprised. This allowed to enhance the text with formatting information.

Listing 2 shows an example of a literate program written in a later, simplified variant of the language (Ramsey 1994). The natural language of the explanation is clearly recognizable. And even if unfamiliar with TeX, the formatting information catches the reader’s eye, like, for example, the expression \section{Printing Primes: An example}, which defines a new heading. The possibility to arrange the Pascal source code becomes likewise evident when looking at the expression <<print the first \([m]\) prime numbers>>, which represents a particular part of the program logic as a human-comprehensible abstraction. It is later substituted – by further abstracting expressions that eventually are substituted by plain Pascal source code\(^{12}\). To obtain an executable program, you can automatically extract the plain Pascal source code from the literate program (see listing 3), which then can subsequently be processed by a Pascal compiler. However, the actual strength of Literate Programming comes to light when, analogously to the Pascal code, extracting the plain TeX part from the literate program. That part consists of the explanation with formatting information, including the chunks. Processed by the appropriate TeX programs for print output, it looks like shown in figure 3.

Certainly, the term literate was justified for this technique. However, Literate Programming as proposed by Knuth was much more than a sophisticated variant of the common practice of commenting source code. The possibility to arrange source code accompanied by human-comprehensible abstractions offered new possibilities related to programming itself. Besides augmenting source code documentation, Knuth found that by applying Literate Programming the quality of his source code had increased:

“My programs are not only explained better than ever before; they also are better programs, because the new methodology encourages me to do a better job.” – (Knuth 1984, p. 97)

In the depicted example, the arrangement of program logic still resembles a tree-like decomposition known from Structured Programming, but literate programs could also own a web-like structure. Thus, Literate Programming facilitated a top-down as well as a bottom-up approach in writing a computer program (Knuth 1984, p. 107).

4.3.2 Discussion

The relation to program comprehension is obviously strong in the case of Literate Programming, because its concept of explaining the program logic considerably supports the forming of a

\(^{12}\)not shown in listing 2
Listing 2: Beginning of a literate program that implements an algorithm for printing the first 1000 prime numbers

\section{Printing Primes: An example}

This program prints a table of the first thousand prime numbers.
We begin by reducing the entire program to its top-level description.

<<
<program to print the first thousand prime numbers>>

\subsection*{This program has no input, because we want to keep it simple.}
The program itself is capable of generating the first \([m]\) prime numbers for any positive \([m]\), as long as the computer’s finite limitations are not exceeded.

<<
<program to print the first thousand prime numbers>>=

program print_primes(output);
\hspace{1cm} constant m = 1000;
\hspace{1cm} var
\hspace{1cm} begin <<print the first \([m]\) prime numbers>>
\hspace{2cm} end.
<<
\subsection*{Plan of the program}

Let us decide at this point to maintain a table that includes all of the prime numbers that will be generated, and to separate the generation problem from the printing problem.

<<
<print the first \([m]\) prime numbers>>=

<<fill table \([p]\) with the first \([m]\) prime numbers>>;
<<print table \([p]\)>>

\subsection*{taken from CTAN (2010) in a revised form}

Listing 3: Complete Pascal source code extracted from the literate program shown in listing 2

program print_primes(output); constant m = 1000; rr = 50; cc = 4; ww = 10; ord_max = 30; { p_ord_max squared must exceed p_m } var p: array [1..m] of integer; { the first m prime numbers, in increasing order } page_number: integer; page_offset: integer; row_offset: integer; c: 0..cc; j: integer; { all primes <= j are in table p } k: 0..m; { this many primes are in table p } j_prime: boolean; ord: 2..ord_max; { the smallest index >= 2 such that p_ord squared > j } square: integer; { square = p_ord squared } n: 2..ord_max; { runs from 2 to ord when testing divisibility } mult: array [2..ord_max] of integer; { runs through multiples of primes } begin j := 1; k := 1; p[1] := 2; ord := 2; square := 9; while k < m do begin repeat j := j + 2; if j = square then begin ord := ord + 1; square := p[ord] * p[ord]; { at this point ord <= k } mult[ord-1] := j; end; n := 2; j_prime := true; while (n < ord) and j_prime do begin while mult[n] < j do mult[n] := mult[n] + p[n] + p[n]; if mult[n] = j then j_prime := false; n := n + 1; end; until j_prime; k := k + 1; p[k] := j; end;

begin page_number := 1; page_offset := 1; while page_offset <= m do begin

begin write('The First '); write(m:1); write(' Prime Numbers --- Page '); write(page_number:1); writeln; writeln; for row_offset := page_offset to page_offset + rr - 1 do begin for c := 0 to cc - 1 do if row_offset + c * rr <= m then write(p[row_offset + c * rr]); writeln; end; page; end;

page_number := page_number + 1; page_offset := page_offset + rr * cc; end; end.

\subsection*{automatically generated according to Ramsey (1994)}
1 Printing Primes: An example

This program prints a table of the first thousand prime numbers. We begin by reducing the entire program to its top-level description.

\[ \langle \{ \text{* to} \} \rangle \equiv \]
\[ \langle \text{program to print the first thousand prime numbers} \text{ \{\text{end}\} } \rangle \]

This program has no input, because we want to keep it simple. We declare the value \( n = 1000 \) as a compile-time constant. The program itself is capable of generating the first \( n \) prime numbers for any positive \( n \), as long as the computer’s finite limitations are not exceeded.

\[ \langle \text{program to print the first thousand prime numbers} \text{ \{\text{end}\} } \rangle \equiv \]
\[ \text{program} \ '\text{print_primes(output)}'; \]
\[ \quad \text{const} \ n = 1000; \]
\[ \quad \langle \text{other constants of the program} \rangle \]
\[ \quad \text{var} \ '\text{variables of the program} \rangle \]
\[ \quad \text{begin} \ '\text{print the first} n \text{prime numbers}'; \]
\[ \quad \text{end}. \]

2 Plan of the program

Let us decide at this point to maintain a table that includes all of the prime numbers that will be generated, and to separate the generation problem from the printing problem.

\[ \langle \text{print the first} n \text{prime numbers} \rangle \equiv \]
\[ \langle \text{fill table} \ p \text{ with the first} n \text{prime numbers} \rangle; \]
\[ \langle \text{print table} \ p \rangle \]

Figure 3: Beginning of the explanation extracted as TeX part from the literate program (listing 2) and subsequently processed for print output\(^{a}\)

\(^{a}\) automatically generated according to Ramsey (1994)
mental model of the program (see section 2.1). Hence, this asset led to further developments of the computer language from Knuth (1983) for creating literate programs\textsuperscript{13} as well as of the concept of Literate Programming itself (Knasmueller 1996, Nørmark 2000). Another benefit of Literate Programming related to program comprehension is the free choice between top-down and bottom-up approaches in writing programs, which might be reflected likewise in a free choice between top-down and bottom-up comprehension strategies. The meeting of the PRIMARILY PROGRAM COMPREHENSION requisite is obvious.

By applying Literate Programming as a documentation technique, the programmer has to focus on the source code. This complies with the LOW LEVEL CONTEXT requisite. In addition, the natural language, which the text is written in, complies with the agile practice of using user stories for written artifacts (see section 2.2.4). And possibly, the HIGH LEVEL BENEFIT requisite is met to a certain degree. This depends on the abstraction level that was employed by the programmer for the explanation and the arrangement of the chunks. A too “low” abstraction level probably cannot deliver an adequate high-level documentation.

Nonetheless, the concept of Literate Programming as well as actually performing it is not a simple concern. Consequently, the agile principle of simplicity (see section 2.2.2) is infringed. Knuth (1984) admitted to the high effort that was necessary to format the text with TeX, the common unfamiliarity of programmers with TeX, and an increase of potentially occurring program errors by combining a programming language with TeX and further language elements. Moreover, the explanation – regarded as documentation – heavily clutters the “actual” program, which according to the second agile value (see section 2.2.1) is the plain source code of the program logic implementation. Changing the documentation after having changed the program logic (or vice versa) would be relatively laborious. Nørmark (2000) solved this problem partially with \textit{Elucidative Programming}, a further development of Literate Programming, by separating program logic and explanation in different files. Still, only due to the aim of documenting, the programmer’s programming activities would be limited to a certain programming concept. This limitation is definitely conflicting with the second agile value. Therefore, Literate Programming and other concepts closely based on it cannot be considered for documenting in an agile context. However, Literate Programming loosely inspired later concepts for documenting source code (Schugerl et al. 2009). One of these, \textit{doc comments}, will subsequently be elaborated as conform to agile values.

### 4.4 Doc comments

#### 4.4.1 Summary

\textit{Doc comments} are a special kind of source code comments, which were originally introduced with \textit{JavaDoc} (Friendly 1995). \textit{JavaDoc} is a tool for automatically generating documentation by means of these doc comments, which are placed within \textit{Java} source code. It was inspired by Literate Programming (see section 4.3). But it fundamentally differed from that by characterizing its concept as “to take the executable program and translate it into a literate description”

\textsuperscript{13}The presented variant of it (Ramsey 1994) is even capable of comprising an arbitrary programming language for program logic implementation. Otherwise, the \texttt{ARBITRARY PROGRAMMING LANGUAGE} requisite would not have been met.
(Friendly 1995, sec. 1.2), whereas Literate Programming followed the opposite direction. Furthermore, its default output format for the documentation was HTML for interactive usage in a web browser instead of processed TeX, which is rather suitable for printout. Friendly emphasized the benefit an HTML documentation had on accessibility and navigation. However, its most distinguishing feature was its focus on solely documenting the API. In the case of Java, OOP as programming paradigm is implied. Thereby, mainly classes and their public methods form an API.

Listing 4 shows an example of Java source code enriched with JavaDoc’s doc comments. There are three doc comments in the example code. These are prefaced with /** instead of /*, as it would be the case for conventional Java comments. Thus, the JavaDoc tool can distinguish doc comments from conventional ones, which it ignores when generating the documentation. As clearly visible, doc comments can be attached to a class as well as to a method. Not shown in the example, yet possible, is to attach them to Java interfaces and variables. Each of these Java language elements an access scope is assigned, and “any items which are private are not part of the API” (Friendly 1995, sec. 2.1). Therefore, those are generally not enriched with doc comments according to the purpose of documenting an externally accessible interface, although it is possible. The doc comments have their own syntax. Besides the plain text, as known from conventional comments, they contain doc tags, which begin with @. The illustrated doc tags from the example have the following meanings:

@version Adds a Version subheading with the specified text to the generated documentation. This tag is intended to hold the current version number of the software that the source code is part of.

@author Adds an Author entry with the specified name(s) to the generated documentation.

@since Adds a Since heading with the specified text to the generated documentation. This tag means that this change or feature has existed since the software version specified by the text.

@return Adds a Returns section with the description text, which describes the return type and permissible range of values. Obviously, this doc tag is valid only in a doc comment for a method.

@throws Adds a Throws subheading to the generated documentation, with a class name of an exception, which may be thrown, and a description text for that. Obviously, this doc tag is valid only in a doc comment for a method.

(Oracle 2010b)

JavaDoc offers further 14 doc tags (Oracle 2010b). One among them, which is commonly used, is the @param tag. It adds to the documentation a name from one of a method’s parameters followed by a description. In addition, HTML code can be embedded within a doc comment. This was implemented to facilitate content-richness, particularly to be capable of adding source code examples – for example, to demonstrate the use of a method (Friendly 1995). Figure 4 shows the complete HTML documentation generated from the commented source code in listing 4. As recognizable, JavaDoc considers further issues related to OOP independently from the stated doc comments, like highlighting constructor methods or retrieving inheritance information that belongs to the class.
Listing 4: Java source code of class java.net.CacheResponse

```java
package java.net;

import java.io.InputStream;
import java.util.Map;
import java.util.List;
import java.io.IOException;

/**
 * Represent channels for retrieving resources from the ResponseCache. Instances of such a class provide an InputStream that returns the entity body, and also a getHeaders() method which returns the associated response headers.
 *
 * @version 1.1, 03/09/22
 * @author Yingxian Wang
 * @since 1.5
 */
public abstract class CacheResponse {

    /**
     * Returns the response headers as a Map.
     * @return An immutable Map from response header field names to lists of field values. The status line has null as its field name.
     * @throws IOException if an I/O error occurs while getting the response headers
     */
    public abstract Map<String, List<String>> getHeaders() throws IOException;

    /**
     * Returns the response body as an InputStream.
     * @return an InputStream from which the response body can be accessed
     * @throws IOException if an I/O error occurs while getting the response body
     */
    public abstract InputStream getBody() throws IOException;
}
```

*a*downloaded from java.net (2006)
Class CacheResponse

java.net.CacheResponse

Direct Known Subclasses:
SecureCacheResponse

public abstract class CacheResponse
extends Object

Represent channels for retrieving resources from the ResponseCache. Instances of such a class provide an InputStream that returns the entity body, and also a getHeaders() method which returns the associated response headers.

Since
1.5

Constructor Summary

CacheResponse()

Method Summary

abstract InputStream getBody() Returns the response body as an InputStream.
abstract Map<String,List<String>> getHeaders() Returns the response headers as a Map.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Constructor Detail

CacheResponse()

Method Detail

getHeaders

public abstract Map<String,List<String>> getHeaders() throws IOException

Returns the response headers as a Map.

>Returns:
An Immutable Map from response header field names to lists of field values. The status line has null as its field name.

>Throws:
IOException: if an I/O error occurs while getting the response headers

getBody

public abstract InputStream getBody() throws IOException

Returns the response body as an InputStream.

(Returns an InputStream from which the response body can be accessed

>Throws:
IOException: if an I/O error occurs while getting the response body

\*\*Figure 4: JavaDoc-generated HTML documentation of Java class java.net.CacheResponse\*

\*captured online from Oracle (2010a)
4.4.2 Discussion

Before deepening the discussion, it is inevitable to mention that JavaDoc violates the ARBITRARY PROGRAMMING LANGUAGE requisite because it was exclusively designed for a use with the Java platform. In fact, that is true, but the emerge of JavaDoc has continuously led to several comparable implementations that operate on other object-oriented programming languages. JavaDoc has taken the position of a de-facto standard in this process (Spinellis 2010). Apparently, a large part of the prevalent programming languages is covered by these derivative implementations (Wikipedia 2010b). Most notably among them is Doxygen (van Heesch 2010), which operates on various programming languages. Hence, JavaDoc – seen as a standard instead of as a tool – yet meets the ARBITRARY PROGRAMMING LANGUAGE requisite.

The particular contribution of JavaDoc to program comprehension is its support of a top-down comprehension strategy due to its focus on interfaces. The doc comments attached to the methods ease the cognitive efforts when following a method call trace. In this context, also a depth-first browsing through the source code is much more facilitated than with less interface-focused documentation concepts. Analogously to the methods, the explicit involvement of classes in JavaDoc in combination with the generated documentation, which is highly browsable regarding classes, contributes significantly to program comprehension: Because the cognitive model (see section 2.1) of the novices among programmers more often depend on objects in program comprehension than that of more experienced programmers (Détiéenne 2001, ch. 5), JavaDoc offers a comprehension approach that addresses programmers of all levels.

As shown, the documentation concept of JavaDoc can be explained concisely with ad-hoc extensions to existing source code comments without much effort. Hence, it meets the agile principle of simplicity (see section 2.2.2). Likewise, the LOW LEVEL CONTEXT requisite is clearly met, because the documentation is directly attached to the source code in the form of doc comments. In opposite to Literate Programming, only interfaces are intended to get documented, which, in OOP, are in substance constituted by public methods. This limitation ensures the relevance of the documentation generated by the JavaDoc tool. A documentation of a method’s implementation is excluded in the documentation concept behind JavaDoc. Hence, the potentially highly optimized – thus hard to understand – source code of the implementation does not influence the comprehension process, as illustrated in the rationale for a documentation-supported program comprehension (section 1.2). In combination with object-oriented design principles that are commonly applied in agile software development (Martin 2003), the limitation to interfaces further strengthens the relevance. Notably among those in this context is the Single-Responsibility Principle, which states that “a class should have only one reason to change” (Martin 2003, p. 95). This leads to a “separation of concerns” (Dijkstra 1982, pp. 60-66) in the source code, which reinforces the relevance of the interfaces and therefore of their documentation generated with the attached doc comments. Further, JavaDoc is obviously promising for a documentation in an agile context by meeting many recommendations for this purpose:

• Not surprisingly, as explicitly mentioned in this pattern, JavaDoc meets the Single Source and Multiple Targets pattern by Rüping (see appendix D) with its ability to create different output formats – as indicated, HTML is only the default one.

• Ambler (2004, ch. 4.10) gave the recommendation to index the documentation, which is definitely met by JavaDoc, as clearly recognizable in the illustrated example (see figure
4). In addition to that, the generated HTML sites are entirely linked to each other. Thus, a requisite for a Document Landscape is met – again a pattern by Rüping (2003), stating that the existence of the available documentation should be conveyed.

- With its @version doc tag, JavaDoc meets the requisite for a documentation history as recommended in a pattern of the same name by Rüping.

- As mentioned, JavaDoc’s orientation to content-richness through embedded HTML facilitates the adding of source code examples, a feature that has explicitly been included in JavaDoc since its original design (Kramer 1999). Likewise, Rüping (see appendix D, Realistic Examples) and Forward & Lethbridge (2002a) suggested to incorporate concrete examples instead of only general explanations in the documentation.

- Among this content-richness you can also number the JavaDoc feature to add introductory explanations as an overview to the documentation. Such overviews are not restricted to Java classes, via doc comments and plain HTML files they can also be attached to Java packages or to the whole generated documentation as an application-wide overview\(^\text{14}\) (Oracle 2010b).

At this point, the discussion of the meeting of the HIGHLEVELBENEFIT requisite is suggesting. Potentially, it is met, because the documentation can be augmented with a high-level complement to the plain interface documentation, which obviously is of a low-level nature. Hinted is the idea of an additional value: Such a high-level complement could at least contain a Big Picture, described by Ambler (2010a, sec. 1) as a “road map” and expressed by Rüping (2003) in the form of a pattern of the same name (see appendix D). However, it remains uncertain to which extent the HIGHLEVELBENEFIT requisite is met. In this respect it is worth mentioning that in an agile context, a simple high-level addition – as a road map – to a presumably huge and very detailed interface documentation seems not properly balanced considering the NOSEPARATEARTIFACT requisite. Furthermore, it should be clear for whom the documentation is designated as any documentation should “have a specific customer” (see also appendix D, Target Reader), and for which purpose as it should “fulfill a single purpose” (Ambler 2004, p. 124). In practice, it is to answer whether the high-level documentation is a source code documentation for developers or an “external” documentation for other stakeholders, and – as elaborated in section 3.2 – whether it is required during development or after that for maintenance (Ambler 2010a, sec. 6).

As a prime example of this difficulty could serve the design process of the API documentation of the Java platform, reported by Kramer (1999): The documentation should have been an API Developer Documentation as well as an API Specification Documentation. The former was exclusively for the application developers, the latter largely for the licensees porting the Java platform. Due to the differing requirements, it was impractical to satisfy both kinds of API documentation to a full extent at the same time. It was decided to concentrate on one kind – the API Specification Documentation\(^\text{15}\). This concrete example points to a very probable violation of the NOSEPARATEARTIFACT requisite as well as of the PRIMARILYPROGRAMCOMPREHENSION requisite: The documentation of the API itself – regardless for which target audience specifically – seems to be the actual goal. Nonetheless, both requisites can potentially be meet,

\(^{14}\)The link to this type of overview is labeled with “Overview” in figure 4.

\(^{15}\)The example shown in figure 4 is part of that.
as it was explained with the example of an API reference in section 3.3. Another common example concerning the target audience is the finding of de Souza et al. (2007) that, in the context of software maintenance, newcomers, apprentices, interns, and experts within a software organization each need different views on the system and thereby different types of documentation. However, the difficulty is not likewise clearly recognizable in this case. Therefore, the API documentation example is referred to below in this thesis for discussions of other techniques or tools instead.

4.5 Unit tests

4.5.1 Summary

The concept of testing software by means of unit tests will not be expounded here. That was already done in section 2.2.3 when elaborating the XP practices which are equivalently relevant for agile software development in general. However, if considered as a documentation technique like here, it should be repeated for emphasis that unit tests are a form of white-box testing (van Deursen 2001, Weyuker 2002) and that they can be seen as an executable work product compared to a static documentation, as noted in the introduction. According to Ambler (2010a), that can even reach the form of an executable specification in the case of TDD. However, as anticipated in section 4.2, this notion is obviously too much influenced by the origin of his agile documentation principles and practices – Agile Modeling – and therefore, it cannot be taken into account in general. Contrary to that, the primary purpose of unit tests apart from their actual testing purpose is to explain the source code of a program (Forward & Lethbridge 2002b, van Deursen 2001). In this context, they are explicitly intended to constitute a documentation, as using them for that is encouraged by XP (van Deursen 2001) – the origin of this testing practice.

Central in the context of unit tests but not mentioned so far is the code coverage. This size denotes that percentage of the source code of a program which has been covered by unit test or, more generally, by any other form of white-box testing. Measuring it can be done by considering different criteria related to the source code. (Zhu et al. 1997) In the case of OOP, it is implicated to consider classes and their methods as such criteria. Then, the full coverage of a class is given if for all its methods unit tests were written.

4.5.2 Discussion

Due to the explanatory nature of the unit tests in respect of the source code, the PRIMARILYPROGRAMCOMPREHENSION requisite evidently is met. However, as elucidated, they can potentially be written with another idea in mind. Hence, the meeting of the PRIMARILYPROGRAMCOMPREHENSION requisite is not warranted. Actually, they minimize the effort for program comprehension processes to a large part because maintaining or changing code is facilitated to be carried out in a trial and error manner (van Deursen 2001). The developers can carefreely and repetitively try to alter the code early before the accomplishment of an appropriate program comprehension process as resulting harm of the code would be discovered immediately by an execution of the unit tests. Anyway, unit tests can not only diminish the need
for program comprehension, they can also aid developers in future program comprehension processes – which are still inevitable:

“[...] hypotheses can be translated into unit tests, which then can be run in order to confirm or refute the hypotheses [...]” – (van Deursen 2001, sec. 4.1)

That means the hypotheses as part of a developer’s cognitive model – probably following a bottom-up comprehension strategy (see section 2.1) – can be eternalized by writing appropriate unit tests and by subsequently adding those to the test suite. Thus, the hypotheses can be “used” again later – also by other developers. This might be particularly beneficial for the collaborative nature of program comprehension. As quoted in section 1.1, Storey (2005) regarded that aspect of program comprehension as insufficiently theoretically substantiated. In any case, collaboration in respect of program comprehension itself is far more than a theory as its emergence can be taken as granted due to the strong team orientation in agile software development.

When regarding the unit tests as documentation, it has to be emphasized that such one does not represent a sole documentation of a program, which comprises all relevant issues of that. Furthermore, those issues that are documented by means of the unit tests are only documented partly, because it is virtually impossible to reach a code coverage of 100% in practice. (van Deursen 2001, Ambler 2010a) As a complement to source code refactoring, as Ambler (2010a, sec. 4) regarded them, unit tests can lead to a more complete documentation. Still, the documentation might not be comprehensive then. Anyway, this is not aimed at all according to the second agile value. And instead of the complementary relation to code refactoring much more characteristic of unit tests is that they are not only code themselves, attached to the code of the actual software in development – what, by the way, fully complies with both the LOW-LEVELCONTEXT requisite and the NOSEPARATEARTIFACT requisite. They are further a part of the developed software product itself. In this context, the documentation formed by the unit tests is at a high probability up-to-date such that those are consistent with the source code of the program to test. The rather self-evident reason for this is the well-built possibility to detect inconsistencies by executing the unit tests and to fix them subsequently. Reported failures or errors in the unit test code itself would then indicate a potentially “obsolete” documentation. In this context, Ambler (2010a) regarded the added value through the unit tests, namely a facilitated validation of the source code of a program, as a motivation for developers to keep their unit tests and thereby the documentation of the program up-to-date.

Considering the HIGHLEVELBENEFIT requisite, it can be indisputably stated that, by adding unit tests as part of the program comprehension process, the requisite is met. Writing unit tests this way increases the code coverage and further the significance of the unit tests in general if they are explicitly (re-)written for program comprehension. At least the former can be regarded as a direct benefit for the customer because a high code coverage definitely indicates a high source code quality. In this case, the system-level documentation necessary for meeting the HIGHLEVELBENEFIT requisite could be the rendering of the code coverage, generated from the unit tests. For that, there exists an abundance of tools (Grigg 2007). Nonetheless, it is open to which extent the code coverage is increased compared to the high code coverage which is prevailing in any case, due to the practice to strive for that. Thus, an increased code coverage seems to be rather a quality of (agile) software development itself. Anyway, this elaboration of unit tests as a documentation for supporting program comprehension will further lead to the subsequent section 4.6, analogously to Literate Programming (section 4.3), which served as
an introduction for doc comments (section 4.4). Besides an increased code coverage, another possibility to achieve a benefit in terms of the **HIGHLEVELBENEFIT** requisite could lie in the extension proposed by Brolund & Ohlrogge (2006) for their own unit test framework **RMock** (Agical 2007):

> “By adding some extra information in a (sub-)suite of test cases and use them to generate the documentation, this information can be structured more human-friendly and complement API documentation such as Java-doc with a more usage/function-oriented view of the system.” – Brolund & Ohlrogge (2006, p. 1)

However, the **NOSEPARATEARTIFACT** requisite would probably be violated by that, likewise to the elucidated violation of it in the case of doc comments. Actually, the tool of Brolund & Ohlrogge (2006) that implements their proposed extension along a test-driven understanding – **TDDoc** – bases on a technique comparable to doc comments. For this reason it is not discussed here. Another aspect of software testing not discussed is the fact that, besides unit tests, also acceptance tests are counted among the techniques which come into consideration for the testing practice from XP (see section 2.2.3). They are not considered in this thesis because the **ARBITRARYPROGRAMMINGLANGUAGE** requisite would be violated in the broader sense: Compared to unit test frameworks, for which a practically standardized conceptual basis – xUnit (Fowler 2007, Meszaros 2007) – exists, there are several different tools that facilitate executable acceptance tests (Park & Maurer 2008). Seen as software development tools, the application of those would bring limitations in respect of generality similar to choosing a specific programming language for development. Apart from that, it is questionable whether the **LOWLEVELCONTEXT** requisite would be met at all, because acceptance tests are a form of black-box testing, even though concrete implementations of them can be interwoven with unit tests (see section 2.2.3).

### 4.6 Scenario diagrams of unit tests

#### 4.6.1 Summary

Test source code was regarded by Forward & Lethbridge (2002b, sec. 3.5) generally as highly valuable for extracting documentation from that. Contrary to section 4.5, where unit tests were elaborated as a form of documentation themselves – including the code coverage as a high-level benefit of that – this section deals with a graphical documentation substantially obtained through such an **extraction** from those: Cornelissen et al. (2007) proposed the idea of obtaining sequence diagrams from test cases – each a unit test or an acceptance test – by reverse-engineering (see section 3.3) them. Such a visualization was explicitly intended by them to serve program comprehension processes in respect of the code tested by the test cases.

A sequence diagram can be seen in figure 5. It shows the interaction between objects, which is considered as being constituted by messages. In a sequence diagram as known from the **Unified Modeling Language** (UML), the objects are framed and horizontally aligned in the diagram, and the interaction in the form of messages is depicted as arrows from one object to another object. The beginning and the ending of each arrow are attached to a so-called lifeline of an object. That is needed for each object because a sequence diagram shows the interaction – vertically aligned – in the form of a sequence of events. The order of the messages, which lead to these events, is
more important than the actual messages themselves. Still, it is distinguished between different types of messages: A message to an object – this might be a method call in the case of OOP – is indicated by a solid line arrow, whereas a return message – for example, a return value from that method – is indicated by a dashed line arrow. There are made further distinctions, which will not be explained here, e.g. the distinction between synchronous and asynchronous messages. (Bell 2004) A set of sequence diagrams can represent the interaction that is characterized by a use case because the sequence diagrams describe the “possible exchanges of messages between objects for various possible scenarios” (Briand 2003, sec. 2). This understanding was shared by Cornelissen et al. (2007), who considered single scenarios for their research and used a simplified version of a sequence diagram for that where necessary. For emphasis, they denote their sequence diagrams consistently scenario diagrams – what will likewise be done here from now on. Further crucial in the context of the notion of a scenario by Cornelissen et al. and their work built up on that is their view on test cases: Each one was considered as a single scenario. Similarly, Ambler (2010c) – in association with his Agile Modeling work (Ambler 2010b) – expressed this view as a “usage scenario”, which could be a part of a use case. Extracting such a scenario from source code and subsequently obtaining a scenario diagram can potentially be accomplished by two kinds of code analysis:

“In obtaining scenario diagrams from testcases, we can choose whether to capture the system’s behavior by means of static analysis (i.e., analyzing the code) or through dynamic analysis (i.e., tracing the execution). The [...] dynamic technique potentially offers more details [...]. A [...] drawback [...] is that one needs specific scenarios to exercise the system. These scenarios, however, come for free when using testcases to drive the comprehension process.” (Cornelissen et al. 2007, sec. 2.1)

This clarifies the specific motivation of Cornelissen et al. for choosing test code instead of the actual code of a program for obtaining scenario diagrams. The mentioned tracing of the execution of the test cases can be realized with different approaches. For that, Cornelissen et al. developed their own event-based framework, in which an event – OOP was assumed – was represented by the beginning or the ending of a method call. Implemented in the form of a prototype tool\(^1\) with Java and its aspect-oriented extension AspectJ (Eclipse 2010a), static

\(^{1}\text{According to Cornelissen (2010), the tool has not been developed beyond its prototype state. However, JRET (Voets 2008) is an applicable implementation of the same technique.}\)
methods and constructor methods were treated separately by the tool. This facilitated the filtering of those in order to reduce the amount of information of the generated scenario diagrams for a better readability – one of the simplifications characteristic of scenario diagrams opposed to sequence diagrams. Further possible simplifications like that were worked out by Cornelissen et al., also possibilities of abstracting information. Most notably among the latter was the limitation to a certain abstraction level, realized through a minimum as well as a maximum depth in respect of tracing an execution call stack. Contrary to the filtering procedures, most of that abstracting was not implemented in the tool to be carried out automatically by that. Apart from technical difficulties that make a fully automatic generation of such simplified sequence diagrams virtually impossible (Cornelissen et al. 2007, Bennett et al. 2008), that would have been also contradictory to the program comprehension process, which according to Cornelissen et al., is coined by tool interaction through the developer who performs this program comprehension process. The presented sequence diagram in figure 5 is such a semi-automatically obtained scenario diagram, i.e. automatic filtering as well as manual abstracting – through the developer’s interaction – was employed. It has been generated from the source code of a test case for JPacman (Geeknet 2010), an implementation of the popular video game Pac-Man (Wikipedia 2010d). The corresponding scenario diagram of the test case obtained exclusively by the automatic filtering of the tool of Cornelissen et al. – without manual abstracting – can be found in appendix B. By using JPacman as a source for a program to comprehend, Cornelissen et al. eventually carried out a case study to prove their hypotheses in relation to program comprehension, in which their tool was likewise comparatively applied by university students.

4.6.2 Discussion

Obviously, the ARBITRARYPROGRAMMINGLANGUAGE requisite is violated by the tool of Cornelissen et al. (2007), as it has been implemented with Java and AspectJ. However, the fundamental technique of the tool principally is not limited to a specific object-oriented programming language like Java, as it does not make use of any non-standard OOP features. Further, aspect-oriented extension like AspectJ for Java, which might be essential for an implementation of the technique, exist for virtually every common OOP language (Wikipedia 2010a). Hence, the violation of the ARBITRARYPROGRAMMINGLANGUAGE requisite can be neglected, similarly to the one in the case of doc comments (section 4.4), even though – contrary to that case – in this case no existing derivations could be found.

According to Forward & Lethbridge (2002b, sec. 7), “visualizing [...] units tests [...] can be of great help in program comprehension.” Likewise, this is evident from the discussion of unit tests in section 4.5.2, where it was found that their explanatory nature is crucial for supporting program comprehension. Thus, a graphical documentation generated by them potentially inherits this characteristic. Furthermore, the above summarized work of Cornelissen et al. was explicitly devoted to program comprehension. Particularly for that they chose to limit to sequence diagrams for visualizing scenarios as these are “easy to read because the chronological order is intuitive” (Cornelissen et al. 2007, sec. 1). Their concept of a dynamical analysis of the source code by tracing its execution for obtaining sequence diagrams was substantiated subsequently by their own research (Cornelissen et al. 2009) as well as by other research works. For example, Bennett et al. (2008) likewise deemed such a concept as supportive for program comprehension:
“ [...] sequence diagrams can also aid understanding of existing software through the visualization of execution call traces.” – (Bennett et al. 2008, sec. 1)

While the PRIMARILY PROGRAM COMPREHENSION requisite, as elaborated, is clearly met, the meeting of the HIGHLEVEL BENEFIT requisite is questionable. There is no similarly recognizable relation to that. However, probably this can be found in a potentially supported change impact analysis, as the most highly prioritized research questions of Cornelissen et al. (2007, sec. 1) was “to facilitate the implementation of change requests.” Eventually, that has been met:

“Our case study has shown that through the application of several abstractions, scenario diagrams effectively visualize how testcases (and the associated functionalities) work, which aids greatly in planning change requests.” – (Cornelissen et al. 2007, sec. 7)

Even though a change request is implemented by a developer, that could be more or less directly related to other stakeholders:

- In the context of program comprehension, finding defects is an important task when performing a change request (Wong 1996). Regarding the object-oriented approach of the UML, which is relatively easy to understand also for non-experts (Détiéenne 2001, ch. 5), and the scenario diagrams, which depict the interaction between those objects, this could possibly also be done partially by a domain expert like, for example, the product manager. A conceivable case could be the potential detection of an omitted object or a missing message to an object. If that object represents a “real” object out of the appropriate domain of the domain expert, that expert might know best how to handle it.

- A project manager who manages the work done by a developer might have to estimate the effort of implementing the change request and the involved risk potential. A change impact analysis, which is commonly seen as a crucial part of appropriate change management processes when planning a change request, is one possible method for that. (Arnold 1996) Actually, techniques for performing a change impact analysis often base on a tracing of execution call stacks similar to that used for obtaining the scenario diagrams (Jashki et al. 2008). As these techniques reveal the interdependencies within the code – among “program entities” – they are also known as dependency analysis techniques (Arnold 1996). Furthermore, such dependency analysis techniques explicitly address OOP as the underlying programming paradigm (Huang & Song 2008) and test code as a candidate for the tracing (Orso et al. 2004). Thus, the tool of Cornelissen et al. (2007) obviously complies with appropriate change impact analysis tools. Apart from developers, also project managers should be involved actively in such a change impact analysis. This is at least suggesting due to the recent observations that non-software artifacts (Jashki et al. 2008, Chen & Chen 2009) as well as business-specific concerns (Oliveira Filho 2010) are commonly neglected in a change impact analysis. Contrary to a developer, a project stakeholder might not be likewise restricted to source code issues.

These stakeholders, despite not being developers, could use the scenario diagrams for their purposes. Consequently, they would have a direct benefit, a potentially improved quality of the developed software product or the progress of software development. Thereby, the HIGHLEVEL BENEFIT requisite would be met. However, it remains an open question whether and how the scenario diagrams could be used in a beneficial way for carrying out a detection of
object-related defects or a change impact analysis by project stakeholders. It is evident that either the scenario diagrams would have to be understandable for them as well, or that it would be necessary to transform those into another form of documentation suited particularly for project stakeholders. Cornelissen et al. (2007) did not state anything about that, as it was by far not related to their research questions. Nonetheless, an adequately direct benefit for other stakeholders than the developers can at least be supposed so far because both a more universal understandability of the diagrams and a possibility of transforming them seem feasible.

As elucidated in the summary, sequence diagrams in terms of scenario diagrams are closely related to use cases. That form of requirements specification – not explained here further – is not as characteristic of agile software development as a user story is, due to its absent adherence to the preferred face-to-face communication (see section 2.2.2). Thus, the relation of the work of Cornelissen et al. to agile software development might not be evident. However, use case and user story can be seen as equivalent in the context of the notion of a scenario as expounded above, although both are usually distinguished from each other very strictly (Cohn 2004, ch. 12). One distinguishing feature according to Cohn (2004, ch. 12) is that a user story is rather not regarded as a permanent artifact. This is identical with the finding of Bennett et al. (2008, sec. 5.2) that sequence diagrams obtained from source code as proposed by Cornelissen et al. (2007) are typically “for personal use, not for sharing with others,” and thus rather a temporary artifact. As explicitly pointed to by Bennett et al. (2008), this might contradict collaboration and thereby appears not to meet agile software development in general. At least, it does not substantially contradict any agile value or principle. Furthermore, the idea of a loosely agile documentation with a limited lifetime was proposed by Forward & Lethbridge (2002a,b) and is feasible as a complement in the form of an “expiration date” to the Notification upon Update pattern (see appendix D) by Rüping (2003). With this understanding, the LOWLEVELCONTEXT requisite is met, which otherwise would be violated as the documentation has to be “attached” to the source code. After the generation of the documentation in the form of the diagrams, this is not anymore the case. Anyway, if that is done individually by a developer who “throws away” the documentation after personal use, then for this developer the documentation is de facto still – cognitively – attached to the source code during its lifetime.

Evidently, the generation of the scenario diagrams is conform to the understanding of reverse engineering as elaborated for introducing the NOSEPARATEARTIFACT requisite in section 3.3. Unfortunately, it comes with the same potential downsides: According to Cornelissen et al. (2007, sec. 1), “if no abstractions are applied, scenario diagrams tend to become too large.” Thus, the manual interaction by the developer seems inevitable regarding the required simplicity of the resulting diagrams. Particularly when considering that even such apparently unimportant issues like the layout of sequence diagrams are crucial in respect of program comprehension (Wong & Sun 2006). This kind of semi-automatic reverse engineering is susceptible to lead to the discussed overdoing of the manual part in favor of the HIGHELEVELBENEFIT requisite. In this case, the NOSEPARATEARTIFACT requisite would be violated. If, contrary to that, the manual interaction were done by the developer solely in favor of a benefit for program comprehension, then the NOSEPARATEARTIFACT requisite would not necessarily be violated and it would rather be a matter of a violation of the lightweight nature of agile software development in general. However, it is conceivable to totally abandon manual interaction as the resulting diagrams would still own a satisfactory value:

“Without any abstractions, three quarters of the unit testcases result in scenario
diagrams that are small enough to be comprehensible.” – (Cornelissen et al. 2007, sec. 5.1)

Nonetheless, it remains open whether there can be reached a benefit in terms of the HIGH-LEVELBENEFIT requisite when resigning the abstractions. After all, Cornelissen et al. (2007) considered only the understandability for developers, not that for project stakeholders. The latter might not possess the same ability to abstract from the diagrams by means of their own cognition. In the context of understandability, it has further to be mentioned that this is the first time an explicitly graphical documentation instead of an textual one is considered. In general, it is unclear which kind of documentation is more suitable for developers (Tilley & Huang 2003) as well as for project stakeholders (Tilley 2009) in respect of software documentation.

Among the term test case, Cornelissen et al. (2007) also counted acceptance tests. Because those were rejected in the discussion of unit tests in section 4.5.2 due to a violation of the ARBITRARYPROGRAMMINGLANGUAGE requisite, they are likewise not considered here for generating documentation. Further, Briand (2003, sec. 2) suggested to use, besides sequence diagrams, statechart diagrams for depicting interaction, particularly that of “classes with state-dependent behavior.” However, considering those would violate the ARBITRARYPROGRAMMINGLANGUAGE requisite in terms of a specific employment of OOP: Contrary to a message-based programming, a state-based programming is not generally characteristic of OOP. This can be deduced from the popular Design Patterns for object-oriented software development by Gamma et al. (1995) – also known as the Gang of Four – which advise the following approach:

“An object performs an operation when it receives a request (or message) from a client. Requests are the only way to get an object to execute an operation. [...] Because of these restrictions, the object’s internal state is said to be encapsulated; it cannot be accessed directly, and its representation is invisible from outside the object.” – (Gamma et al. 1995, p. 11)

Moreover, according to Bennett et al. (2008), an analysis of such classes for obtaining diagrams that depict interaction can only be performed on an object level, i.e. concrete instantiations of these classes. Such an object-level analysis brings certain problems regarding understandability, which make the thus obtained statechart diagrams inappropriate for further considerations:

“[...] being able to distinguish between objects provides detailed information on object interactions [...]. However, with diagrams at the class level already being susceptible to size problems, one will definitely encounter scalability issues with additional object information.” – (Cornelissen et al. 2007, sec. 2.1)

4.7 Links to user stories

4.7.1 Summary

Ratanotayanon et al. (2009) proposed an innovative concept for supporting program comprehension that facilitates the traceability of user stories on the source code level. They have worked out this in the form of a visualization technique based on links from component parts
of a source code to user stories. These links and their visualization were intended to help a developer to locate conceptionally related sections of the code. The goal of Ratanotayanon et al. was to map a high-level description of a feature or task to such code sections, in a way that fully complies with agile software development in general. For that, they implemented a prototype tool – Zelda\textsuperscript{17} – a plug-in for the integrated development environment (IDE) Eclipse (Eclipse 2010a). Originally, it has been dedicated to be used with Java but is principally – except for one minor function – usable with any other programming language supported by Eclipse. The links are visualized by it in five fundamentally differing ways, also called views. These are listed here from a close adherence to user stories to a close adherence to the source code, and, secondarily, from a high to a low abstraction level:

- A list of linked user stories
- Marking of the files that contain links or that are themselves as a whole linked to user stories in the explorer view of Eclipse
- A graphical overview of those files, each depicted by a “block” including the linked parts in the form of “stripes” within the block (see figure 6a)
- A hierarchically structured list of the code elements that contain links to user stories, according to the program structure – for example, in the concrete case of Java, ordered by package, class, and method (see figure 6b)\textsuperscript{18}
- Marking of the code lines that are linked to user stories in the code editor of Eclipse

All these views are appropriately navigable. For example, a click on any of the markers leads to the linked user story. Such a linking in general is not characteristic of the tool of Ratanotayanon et al. There are other properties of their developed tool that makes it, according to their own statements, unique compared to other similar tools. These properties are:

- The traceability was not pursued by considering – as usual in that research field – requirements specifications, architecture documents, or design models. This made the resulting tool conform to agile software development because those kinds of documents are not common for that.
- Not only source code files can be linked to user stories but also entities of it on another abstraction level, particularly code lines.
- Explicitly, also files with code of test cases as well as “non-code” files – or the component parts of those – can be linked to user stories.
- A revision control system (RCS), which is commonly employed in software development, can be used to implement an automatic correction of the location of the links in the code when the code – under revision control – was changed.

Eventually, Ratanotayanon et al. carried out a comparative case study similar in design to that of Cornelissen et al. (2007) (see section 4.6.1). In this context, it is most notable that the developers who used their tool were “only marginally faster” but “were more focused, and were more willing to make use of additional information” (Ratanotayanon et al. 2009, p. 31).

\textsuperscript{17}The tool Zelda has been developed further in the meantime, now known as Lilly (Ratanotayanon 2010), available on the website of the Galen Lab of the University of California, Irvine (Galen 2010).

\textsuperscript{18}This is the mentioned minor function not supported for other programming languages than Java.
4.7.2 Discussion

The tool of Ratanotayanon et al. (2009) should explicitly serve the purpose of supporting program comprehension. And its linking technique is clearly attached to the source code, technically as well as cognitively – for the developer. Hence, the primarily program comprehension requisite and the low-level context requisite seem to be met. However, in the case of the latter, it must be added that, even though the support of program comprehension is quite obvious, it is open whether the tool serves primarily that purpose. User stories can be seen as high-level descriptions of a program – or more precisely, of those parts of it which are linked to them – only as long as they were seen as a form of requirements artifact, binding to some degree during the implementation of the program. Then, it can be taken as granted at a high probability that they meet the actual implementation and that thereby they explain – at least vaguely – the source code of that implementation – what substantially supports a program comprehension process, as elaborated in the preceding discussions.

Besides the explicitly stated designation of the tool for supporting program comprehension, it was, in addition, likewise designated to be used in agile software development. Actually, among the found techniques and tools, the proposal of Ratanotayanon et al. is singular due to its equal consideration of both program comprehension and agile software development. In respect of the latter, they found the following problem:

“[…] we cannot apply traditional traceability techniques, as there are usually no documents from which to trace.” – (Ratanotayanon et al. 2009, p. 26)
The only kind of document that came into question for regarding traceability was the user story. As more or less implied in the quoted statement, that is typically not seen as a document, it is rather seen as a “placeholder” for conversation (Cohn 2004). In any case, Ratanotayanon et al. assumed that in agile software development user stories are the prevailing instantiations of that placeholder. Contrary to unit tests, for which no obvious reason exists not to write them in agile software development – they are even deemed as a common practice in software development in general (Ellims et al. 2004, Runeson 2006) – user stories are more interchangeable with other conceivable documents. As introduced in section 2.2.3, they might be the most common kind of document for planning an increment, still not the sole one. This becomes clear when considering the multitude of other kinds of documents in the context of Agile Modeling by Ambler (2010b). Consequently, this results in a constraint of the ARBITRARYAGILEMETHODODLOGY requisite, even though that is basically met. A further constraint is, as stated, that each user story is seen as a high-level document in terms of a requirements specification artifact. It has to be pointed out that not these documents themselves are considered as the documentation for supporting program comprehesnion but the links to them. In this context, the HIGHLEVELBENEFIT requisite, contrary to the ARBITRARYAGILEMETHODODLOGY requisite, is met without any constraints. Actually, both elaborated types of benefits (see section 3.2) are met:

- As expounded in the deduction of the HIGHLEVELBENEFIT requisite, a documentation that supports program comprehension most likely results indirectly in a higher quality of the developed software. This is substantiated by the findings by Ratanotayanon et al. that “implementing” user stories through developers is facilitated by the links to those. Furthermore, the claim of Ratanotayanon et al. (2009, p. 31) that the links “also provide a means to reuse existing knowledge by presenting examples of code usage and strategies for implementing a feature” is consistent with the understanding of unit tests. This can be regarded as another benefit of the second type that is induced here. The high-level documentation necessary for convincing a project stakeholder of both these benefits of the second type could comprise some of the visualizations generated by the tool. At least the visualization consisting of “blocks” and “stripes” (see figure 6a) seems suitable for accomplishing that in the context of a view abstracted from the source code and for being intuitively understandable also for project stakeholders at the same time.

- The suggested visualization as a representation of the documentation on the system-level tempts to use it further as a mean to meet the first type of benefit in terms of the HIGHLEVELBENEFIT requisite: Again, as deduced in section 4.6.2, a change impact analysis could potentially be supported by it. Even though not typically regarded as permanent artifacts (Cohn 2004, ch. 12), already “implemented” user stories in combination with the links to them in the source code could help to estimate the impact that a change request – usually related to these previous user stories (Breitman & Leite 2002) – would have on the code. This characterizes a traceability analysis, a specific instantiation of a change impact analysis (Arnold 1996), likewise to the dependency analysis exemplified in section 4.6.2. Moreover, Ryder & Tip (2001) emphasized that, in the context of OOP, a change impact analysis deals more with the “semantic” impact on source code than the “syntactic” one. It is evident that the user stories and the links to them facilitates that understanding by enriching the code with semantic information in terms of an abstraction from the code. Further, the hierarchically structured visualization (see figure 6b) might be helpful in assessing the “granularity” of the change according to Buckley et al. (2005),
in order to estimate the quality of its impact. This can be deduced from the congruence of the code elements depicted in the visualization with the object-oriented levels they regarded as constituting for their notion of granularity in the context of OOP. While, so far, the facilitation of a change impact analysis by the visualizations in general was exemplified, still outstanding is its reference to project stakeholders. A concrete scenario for an involvement of project stakeholders in the change impact analysis could be the case that they estimate the impact on the basis of the number of the affected files. These are recognizable, for example, as marked files in the explorer view of Eclipse as extended by the tool. Thus, this visualization could be regarded as an appropriate high-level documentation, because it seems to be understandable also for project stakeholders in respect of the elaborated purpose. Even though the number of files affected by a change is not very meaningful for estimating its quantitative impact in the case of OOP (Ryder & Tip 2001), it could be meaningful for a subsequent dependency analysis, then restricted on these files.

Achieving the meeting of the as-yet addressed requisites virtually excludes a violation of the NoSeparateArtifact requisite. Linking files or parts from them to user stories can hardly be considered as a significant additional effort. Particularly because, in the case of an employed RCS, you can let update the links by an automatic correction of their location in the code after a change to that. But also without this possibility the NoSeparateArtifact requisite is met. Otherwise, the ArbitraryProgrammingLanguage requisite would be violated in the broader sense due to the prescription of using a RCS, similarly to its violation by acceptance tests discussed in section 4.5.2. Certainly, the tool of Ratanotayanon et al. itself violates the ArbitraryProgrammingLanguage requisite because it is limited on the programming languages supported by a particular IDE – Eclipse. But the fundamental technique of it including the idea of facilitating an automatic update of the links by means of a RCS could principally be implemented within another IDE. Overall, this technique is relatively lightweight, what is seen as an inevitable requirement for a documentation intended to be used after the initial development of a software product:

“Finally, during development you likely want less documentation, you prefer to travel light, than you do during post-development.” – (Ambler 2010a, sec. 6)

In the context of documentation, traveling light means to create “just enough documentation to get by” (Ambler 2010a, sec. 7) and to subsequently “update only when it hurts” (Ambler 2010a, sec. 12). This advice comes with the compromise of using a documentation that is potentially not (anymore) up-to-date. In this case, Ambler (2010a) is in accordance to the finding of Forward & Lethbridge (2002a, sec. 3.2) that ”out-dated documentation is still quite useful.” However, the idea by Ratanotayanon et al. of automatically updating the links – thus, the documentation – makes these concerns largely superfluous. Further, this idea was explicitly intended by them “to ensure the long-term value” (Ratanotayanon et al. 2009, p. 31) of the links. Hence, their proposed tool and the technique of it conform even more to the motivation for this thesis – besides the above-mentioned explicit designation of their research to program comprehension and, in equal measure, to agile software development.
4.8 Design rationales

4.8.1 Summary

Well-known in the context of software documentation is a documentation based on design decision rationales or short, design rationales (DR). A DR states the reasons behind a decision that was made during the design of a software system as well as the justifications for those. A further view on it is that it deals with the evaluated trade-offs and the argumentation which led to this decision. (Lee 1997) Thus, a DR and a DR documentation are evidently related to software development (Dutoit et al. 2006). Particularly, such a documentation was explicitly referred to agile software development in research works by Lee et al. (2003), Sauer (2003), Falessi et al. (2006). Furthermore, DR in general were appraised highly relevant in agile software development by Martin (2003) as part of his reflection on the second agile value, by Rüping (2003) as one of his patterns (see appendix D), and by Ambler (2010a, sec. 8) in one of his recommendations. While the latter research work of Falessi et al. (2006) considered DR documentation as principally beneficial for agile, the former two research works proposed each a concrete tool for the employment of DR in agile software development, leading to a more or less comprehensive DR documentation. The tool of Lee et al. (2003) has been implemented as a plug-in – Echo – for the IDE Eclipse (Eclipse 2010b), whereas the tool of Sauer (2003) has been implemented as an extension of MILOS (Maurer et al. 2000), which is a tool for distributed software development facilitating web-based project planning and workflow management.

There exists no prescribed form of a DR. It can have different representations, mostly categorized as informal, semi-formal, and formal. An informal representation could be, for example, a description in natural language, whereas a semi-formal representation has to be at least partly computer-readable. For example, this could be met by a filled out template. A completely computer-readable representation distinguishes a formal representation, which can be seen as a knowledge base written in a formal language. (Lee 1997) According to this categorization, the representation of a DR by Lee et al. (2003) can be regarded as informal: A DR can be attached to a requirements artifact. Among these artifacts Lee et al. counted, for example, user stories and feature descriptions. For the latter, figure 7a shows an example. In contrast to that, the representation of a DR by Sauer can be regarded as formal, as it was built up on an event-based ontology:

“Generally spoken, software development projects can be described using a series of planning and enactment events. [...] Events may be created automatically by the tools used during development, e.g., bug-tracking systems, versioning tools, or workflow management utilities.” – (Sauer 2003, sec. 2.1)

Figure 7b shows an example of two DR in the extended MILOS tool. The one automatically created by it due to the occurrence of the appendant event, the other created manually. According to Lee (1997), DR can be created with different approaches, which can be distinguished by the degree to which the tool itself acts in respect of creating the DR: The more formal the representation of a DR, the more the autonomous involvement of the tool is facilitated. This is also evident from the exemplification of the DR representation of Sauer. The approaches for creating a DR can reach from a manual reconstruction of it after the development of the software system to the immediate and completely automatic generation of it by solely the tool itself.
Figure 7: Representations of a design rationale

(a) A design rationale in *Echo*\(^a\)

(b) Two design rationales in *MILOS*\(^b\)

\(^a\) taken from Lee et al. (2003, sec. 4.4)  
\(^b\) taken from Sauer (2003, sec. 3)
then acting as a software agent (Lee 1997). Likewise to the various approaches of creating a DR, the context of it can vary. Commonly, this context is the software system and not the software product. Hence, a DR documentation often relates to software architecture (Falessi et al. 2008). Further, this is implicated when considering that software architecture, according to the *Software Engineering Body of Knowledge* (SWEBOK) by IEEE (2004), is part of software design. However, as Sauer (2003, sec. 2) mentioned, a DR “is not limited to design processes but applies to all phases in software development.”

### 4.8.2 Discussion

Apparently, none of the two presented tools has something to do with program comprehension: Sauer (2003) uses throughout his research work a process-focused example – the assignment of a human resource to a task depending on its previous assignments. And the examples of Lee et al. (2003) are focused on requirements artifacts from a user perspective. However, both tools were supposed to be compatible with agile software development. This starting point makes them eligible for a consideration in this thesis, contrary to tools that might be promising for producing a DR-based documentation in general but that are neither related to program comprehension nor to agile software development in particular. The tool of Burge & Brown (2008) is an example of those. It is not considered further here but only mentioned. Eventually, it seems eligible for a consideration in practice – perchance even for the intended purpose of supporting program comprehension in a way more or less conform to agile software development.

Now, the relation of the two considered tools to program comprehension is to work out: Sauer emphasized that a DR, because it is commonly documented immediately after the appropriate decision making, *explains* the decision rather than defends that. Therefore, a DR in general provides an unaltered, more original view on decisions regarding to the point of time they were made. Likewise to the explanatory nature of the previously elaborated techniques and tools, this is most likely supportive for program comprehension in respect of a subsequently done implementation, which is the source code. Further, as suggested by Sauer, a DR can go beyond the design phase. In addition to a design (decision) rationale, an understanding of a DR as an “implementation (decision) rationale” is conceivable. In such a case, a DR would be more or less exclusively relevant to program comprehension. Consequently, the *PRIMARILY PROGRAM COMPREHENSION* requisite would clearly be met. If the exemplified case of an obfuscating code occurs (see section 1.2), a DR documentation – as a complementary documentation – could give reasons why the implementation – understood as the actual documentation – was done so “unclean” at a certain part of the code. Thereby, that part of the code could become more comprehensible. As long as it can be anticipated with adequate safety that this part will be changed eventually and that more effort is required for the change without such a DR than for the change with such a DR including the initial creation of it, then the second agile value is not infringed by that. Certainly, also some of the other presented techniques or tools like, for example, doc comments (see section 4.4) could be used in a similar manner to clarify the obfuscation induced by the code. But contrary to them, it is *explicitly* intended to document a justification by a DR.

A DR documentation consisting of such DR that refer to the implementation would further meet the *LOW LEVEL CONTEXT* requisite. Nonetheless, both presented tools do not comply with the necessity of providing a documentation attached to the code: A DR from the tool of
Sauer as well as the MILOS tool itself are not associated with the code. Indeed, the tool of Lee et al. is embedded in the IDE, but its actual purpose, according to the research questions of Lee et al., is to facilitate the traceability and stepwise refinement of requirements artifacts. Therefore, it is not warranted that these artifacts – among them the DR – are attached to the code. And if they were, it would be the IDE Eclipse itself and not the plug-in which enable that. Anyway, this is again a violation of a requisite by a tool which can be eliminated by implementing the fundamental technique of that in a way conform to this requisite. Feasible is to link the DR like the user stories are linked by the tool of Ratanotayanon et al. (2009) presented in section 4.7. Further, even a combination of both is at least conceivable. In the context of an alternative implementation of the fundamental technique of the two tools, it has to be pointed out that both tools violate the ARBITRARYPROGRAMMINGLANGUAGE requisite in the broader sense – as most the tools presented above do. However, in the case of the tool of Lee et al., an alternative implementation that conforms, besides to the LOWLEVELCONTEXT requisite, also to the ARBITRARYPROGRAMMINGLANGUAGE requisite, is not only conceivable but was originally intended by Lee et al.:

“However, it has been designed to easily port to other [...] development environments, such as NetBeans, Together Control Center, etc.” – (Lee et al. 2003, sec. 4.3)

Because the research work of Sauer (2003, sec. 2) “focuses on the documentation aspect on DR in general and how to benefit from it at as little cost as possible,” it is evident that his tool was considered to induce a form of documentation. As shown, this can support program comprehension processes and, as anticipated by the emphasis on agile software development by Sauer, is at the same time suitable for an employment in that. The same applies to the tool of Lee et al. (2003). Besides, the general compatibility of DR with agile software development – due to the possibility that these DR own a lightweight nature – is further confirmed by Lee (1997), who explicitly stated that DR could be used for an automatic generation of documentation. Surely, that highly conforms to agile software development in general. However, a prerequisite for that is the formalization of knowledge, which is costly in respect of the effort of creating a documentation (Lee 1997). Because of the high effort for such a documentation, it is not widely used in practice, despite its obvious benefits (Falessi et al. 2008). Contrary to this costliness – generally anticipated when a DR documentation is generated automatically – the tool of Sauer is little costly. Thus, it appears that this tool is not an impediment to agile software development. However, the resulting DR representation does not directly conform to the sixth agile principle (see section 2.2.2), as it is hardly imaginable how such a formal documentation artifact (see figure 7b) can be employed in a face-to-face communication. Unlike this tool, the tool of Lee et al. (2003) evidently conforms to that agile principle. Obviously, the DR representation in the form of a continuous text written in natural language (see figure 7a) is more suitable for an employment in agile software development. Despite the necessary – very costly – manual intervention to create DR, the tool could still entirely conform to agile software development. A concrete scenario for that was elucidated above. Apart from the costs for formalizing a DR documentation, there is a likewise critical issue, namely its target audience. The target audience implicates a violation the NOSEPARATEARTIFACT requisite – principally met elsewise – if the following scenario regarding techniques for a DR documentation comes true:

“Several [...] techniques already exist; however they are usually focused on maximizing the consumer benefits rather than minimizing the producer effort. Con-
sequently, people involved in the documentation and maintenance activities are supposed to spend a huge amount of effort.” – (Falessi et al. 2008, p. 190)

Besides the elaborated compatibility of the tools and the resulting DR documentation with agile software development, there are properties of a DR documentation that reveal also the particular contribution of it to agile software development: Lee (1997), Falessi et al. (2006) considered a DR documentation as helpful for collaboration, what more or less directly meets a large part of the agile principles and practices (see section 2.2.4). According to Falessi et al. (2006), a DR documentation is particularly helpful for collaboration because it can increase the effectiveness of team decision making through conveying design alternatives. Lee (1997) deemed a DR documentation as a kind of “project memory,” which could partially be a remedy for the knowledge decay pointed out in section 1. Thus, a DR documentation as a documentation for the purpose of supporting program comprehension strongly complies with the motivation for this thesis. Likewise to collaboration, communication is characteristic of agile software development. And DR are particularly helpful also for this characteristic according to Shipman & McCall (1997), Horner & Atwood (2006):

“For some people, DR means capturing and retrieving naturally occurring communication – e.g., design discourse – among members of a project team.” – (Shipman & McCall 1997, sec. 1.2)

“One goal of design rationale is to [...] communicate the argumentation and reasoning behind the design process.” – (Horner & Atwood 2006, p. 341)

In the context of communication, it is worth mentioning that Horner & Atwood provided an exemplary suggestion for how to obtain DR that are adequately informal to facilitate – as requirements artifacts – further face-to-face communication: They suggested to integrate “design rationale tools into web browsers, e-mail clients, phone systems, instant messaging tools, and meeting support tools” (Horner & Atwood 2006, p. 348). Moreover, Horner & Atwood claimed that these possibilities could further supports collaboration particularly on a long-term basis. This due to a practically facilitated communication between designers from one point in time and designers from another point in time. Again, this strongly complies with the motivation for the thesis. Besides supporting collaboration and communication, a DR documentation seems also helpful for achieving a benefit in terms of the HIGHLEVELBENEFIT requisite:

“[..] documents help others besides designers. Managers or users can use them to evaluate the design. Lawyers can use them to determine if the design is intellectual property.” – (Lee 1997, p. 79)

Such a benefit would be a direct benefit for a project stakeholder. Hence, the HIGHLEVELBENEFIT requisite is basically met. Another benefit – likewise evidently of the second type (see section 3.2) – can be deduced from the reasoning by Komiya (1994):

“The information recorded [...] could also be referred to as expert knowledge related to software design. If this information can be reused in the initial design phase of new software systems as well as in the software maintenance phase, we can expect an increase in software development productivity and quality.” – (Komiya 1994, p. 200)
Certainly, this exemplified benefit might occur to a certain degree as an indirect consequence of the program comprehension process, which is, as expounded, supported by a DR documentation, and which is surely needed in such a software maintenance phase. However, there is a concrete case that leads independently from program comprehension to this benefit. Similarly to the discussions of the other techniques and tools, it is conceivable that a change impact analysis is part of the maintenance here. This change impact analysis is, again, definitely supported by the fundamental technique of a presented tool – here the DR that constitute a documentation – as well as by the tool itself. The latter is proven at least for the tool of Sauer (2003), who explicitly deemed DR as beneficial for estimating a change impact. To exemplify that, he gave the DR example “priority of process ’design database schema’ set to ’high’ because it produces a prerequisite artifact for every other design process” (Sauer 2003, sec. 2). This example suggested that a change impact analysis basing on dependencies between DR is facilitated. In the course of pointing out DR documentation as beneficial for maintenance, Lee (1997) proposed a similar thought (“dependency management”), which can be seen as a facilitation of a DR-based dependency analysis (see section 4.6.2) likewise to that of Sauer.
5 Survey studies

This empirical part of the thesis builds on the findings from the literature review in section 4 and comprises the presentation and the analysis of the results of a series of survey studies: Three of the six summarized and discussed techniques and tools were chosen to be evaluated in a software organization. This evaluation was carried out with an appraisal by selected professionals of that organization with the purpose to verify the techniques and tools regarding the HIGH-LEVELBENEFIT requisite (see section 3.2). Besides the question whether the requisite could be met, it was investigated how it could be met. Choosing those three techniques or tools and focusing on the HIGH-LEVELBENEFIT requisite is argued in section 5.1. This is followed by the research setting (section 5.2) including a brief presentation of the software organization (section 5.2.1) and the sampling of the target population (section 5.2.2). Subsequently, the research strategy – surveys – including the data collection method – questionnaires – is elaborated in section 5.3 as a form of “soft evaluation” known by Kitchenham & Pfleeger (2008). Because a series of studies was done – one per technique or tool – the unvarying procedures for conducting such a study as well as its structure are explained (section 5.4) before dealing with the studies themselves (sections 5.5 to 5.8) to minimize repetitiveness. The latter should be read one after the other. Especially because the first of the four studies (section 5.5) is an exception: It served not the evaluation of a technique or tool but that of the research framework itself. Furthermore, it allowed to prove the conclusiveness of the results of the subsequent studies.

5.1 Object of research

After the literature review, it is obvious that the effort of documenting for program comprehension is one of the most crucial aspects when considering the conformity with agile software development. Unfortunately, empirical studies for investigating that aspect would require a productive employment of a technique or tool, what would have been too costly in the case of an industrial software organization.\footnote{Another aspect that would definitely require a productive employment for its investigation is the role of interactivity. At least, that aspect was considered crucial by Bennett et al. (2008) for the technique of generating sequence diagrams by tracing execution call stacks (see section 4.6).} However, like the support of program comprehension, the conformity of the presented techniques and tools with agile software development was (dis)proved either through reasoning in the discussions of them or through the original research works from which they were extracted.

Considering the requisites from the research framework, it can be stated that a verification of the techniques and tools in respect of them was likewise accomplished to a large extent. Certainly, a complementary verification by empirical studies could substantiate that, and it would not necessarily be too costly. However, it would not have been expedient for every requisite: The ARBITRARYAGILEMETHODLOGY requisite and the ARBITRARYPROGRAMMINGLANGUAGE requisite are definitely not of relevance for a specific software organization, because such usually does not apply arbitrary agile software development methodologies and arbitrary programming languages but predetermined ones. Contrary to that, the LOWLEVELCONTEXT requisite, the NOSEPARATEARTIFACT requisite, and the PRIMARILYPROGRAMCOMPREHENSION requisite might be of relevance. Evidently, these requisites strongly address software...
developers. That makes them unsuitable for an investigation regarding stakeholders of different professions. Therefore, it was decided to focus on the only remaining requisite – the HIGH-LEVELBENEFIT requisite. That also owns the highest demand for exploratory research among all requisites as no concrete benefit could be identified definitely. An elicitation of such as well as a confirmation of the anticipated facilitation of a change impact analysis was supposed to be achieved by the survey studies.

The evaluation was not only limited to three candidates for a research result but it were further considered only techniques. In the case of tools, this meant to abstract each of those to the sole fundamental technique implemented by that (see section 1.5). Choosing the techniques (or tools) was done by excluding the following ones:

- Literate programming (see section 4.3) does not meet all requisites.
- Doc comments (see section 4.4) cannot be seen as an innovative technique. However, this attribute is a desirable research goal (see section 1.3).
- Unit tests (see section 4.5) were found to induce a direct benefit in terms of the HIGH-LEVELBENEFIT requisite. However, this benefit was also found to be rather an attribute of agile software development than of the particular notion of unit tests in respect of their potential support of program comprehension. Besides, an investigation of that benefit would require, again, a long-term study.

Consequently, only three techniques were left for an evaluation: Scenario diagrams generated from unit tests (see section 4.6), links from source code to user stories (see section 4.7), and design rationales (see section 4.8).

5.2 Research setting

This subsection explains why the software organization HolidayCheck AG was chosen for conducting the studies as well as what the target population within that organization has been and how the sampling of it was done. The latter is expounded separately (see section 5.2.2), as the population is often not well-defined in empirical research in software engineering (Kitchenham et al. 2002, Sjøberg et al. 2007). First, the decision to conduct the empirical studies generally in a software organization is suggesting in the case a maximum comprehensive research setting is desired in terms of a variety of professions. A limitation to one single organization is to prefer to a consideration of several organizations if the collaboration of the stakeholders of the different professions is of higher interest than an enhanced reliability of the results. This is the case here, because the application of agile software development might differ considerably from organization to organization (see section 3.5), and because collaboration was identified as crucial in every discussion of the three chosen techniques and tools (see sections 4.6 to 4.8). Further, due to the following deductions, it was decided beforehand to choose a software organization that (1.) is an industrial one, that (2.) can be counted among small and medium enterprises (SME), and that (3.) develops web applications:

1. Agile software development is known as a “practitioners’ answer to the methods promoted by software engineering research” (Hansson et al. 2006, p. 1296). Thus, it
obviously is much more connected with industry than with academia. Its connection with industry becomes more obvious when considering related software development approaches with an explicit industrial heritage like, for example, lean software development (Poppendieck & Poppendieck 2003). Besides, choosing an industrial software organization for conducting empirical research further increases the relevance of a research work, as empirical research in software engineering is still done too much solely in academia (Sjøberg et al. 2007) and thus not in the setting where its results are actually applied.

2. Mainly SME apply agile software development – proved recently by VersionOne (2010). An explanation for that is the conformity of common attributes of projects carried out by an agile software development methodology with the characteristics of SME. In this context, Koehnemann & Coats (2009, p. 295) stated: “Such projects are small (5-50 members) and have co-located teams working on projects that require a relatively narrow set of skills (web, database). Members are part of the same management structure and are easily redirected based on project needs. These projects are relatively free of governance and reporting such as long-range cost estimates, charge numbers, earned value, and the like.”

3. Typically, agile software development methodologies are increasingly applied for developing web applications today (Souza & Falbo 2005). Particularly due to the need to quickly adapt to changing requirements, which are prevalent in the case of the development of web applications traditionally coming with a time-to-market pressure (Aoyama 1998). VersionOne (2010) revealed the time-to-market aspect as the most common reason for adopting an agile software development methodology at all. Furthermore, Internet companies are typically SME (Altarawneh & El Shiekh 2008, Clutterbuck et al. 2009). This accompanies the deduction to consider solely SME for choosing a software organization. Consequently, it is ensured to have chosen a typical SME organization by a further limitation to Internet companies. Actually, the motivation for the thesis (see section 1.2) has been confirmed by Altarawneh & El Shiekh (2008, p. 128) in this context as they found for SME that develop web applications: “Agile process critics point out that the emphasis on code can lead to corporate memory loss because there is little emphasis on producing good documentation and models to support software creation and evolution of web systems.”

The finally chosen software organization is an Internet company which has been known personally by the author through his previous professional experience (see section 1.7). That organization meets all criteria above. It will be presented in the subsequent subsection. Finally, it should be remarked that not every object of research is relevant for an industrial software organization itself, which might lead to a conflict of interest:

“Nevertheless, collaboration with industry may sometimes lead to studies that have too many interfering factors for relationships to be understood or theories to be built, or to researchers being forced to focus on highly specific and short-term usefulness instead of more general and long-term results.” – (Sjøberg et al. 2007, sec. 5.2)

To prevent such a conflict of interest, two recommendations by Sjøberg et al. (2007) were realized. First, it was made a clear distinction between the research goals of this thesis and the goals of the software organization by pointing out that no “consultancy work” is done. Second,
the participation in the studies took place during the usual working hours of the participants. Consequently, it was paid.

5.2.1 HolidayCheck AG

The information stated in this presentation of the software organization HolidayCheck AG was obtained through an informal interview with the CIO and subsequent ad-hoc interviews with senior developers of that organization. Furthermore, numeric data – status as of 3 November 2010 – was extracted from an internal knowledge data base and the public “About Us” page of the organization’s website (HolidayCheck 2010a).

HolidayCheck AG is an Internet company situated in Bottighofen, Switzerland. It was founded as a start-up in 2003 and still owns a flat hierarchy. Overall, 172 employees excluding trainees and freelancers are employed, most of whom have been recruited directly from university. Productively involved in the software development at HolidayCheck are software developers, project managers, and product managers. There are 25 developers, 12 product managers including cross-product experts for search engine optimization (SEO) and usability, and 4 project managers. HolidayCheck develops e-commerce applications in the tourism branch. Its eponymous main product is a hotel review portal (HolidayCheck 2010b), which is available in twelve countries and ten languages. Most of the company’s revenue comes from the German-speaking version, which has approximately 395,000 unique visits (Wikipedia 2010f) per day, whereas all other versions, summed up, have approximately 55,000 unique visits per day. The characteristics of the portal can be best described by this quotation:

“HolidayCheck’s main areas of competence is the hotel reviews database, the extensive number of private holiday pictures submitted by travellers, as well as a collection of private travel videos. […] HolidayCheck.de has its own online travel agency and collaborates with the most renowned travel agencies.” – (HolidayCheck 2010a)

The portal as well as all other software products are developed with PHP and Java. Further, permanently employed software development tools are the IDE Eclipse, the issue tracking and project tracking system JIRA (Atlassian 2010), and the RCS Mercurial (Babenhauserheide 2010). Due to an enormous growth of the company since its existence, there had been no defined software development process for years. Thus, software development was carried out ad-hoc – typical of small Internet companies (Altarawneh & El Shiekh 2008). Then HolidayCheck started first to adopt several XP practices (see section 2.2.3) before ultimately adopting Scrumban (see section 2.2.5) as a methodology in May 2010. In the company’s implementation of that, the product managers fulfill the role of the product owner, whereas the project managers fulfill the role of the Scrum Master apart from their Scrum-independent project management tasks.20 Regarding the last two requisites of the research framework, having chosen HolidayCheck for conducting the survey studies does not contradict generality:

20The latter is not unusual, as the typical software development life-cycle is not covered entirely by Scrum (Abrahamsson et al. 2003) despite its strong focus on project management in general (see section 3.5).
• In the case of the ArbitraryAgileMethodology requisite, it can be found that Scrum is by far the most often applied agile software development methodology in industrial settings (Begel & Nagappan 2007, VersionOne 2010), with Scrumban as a recent trend (Tørresdal 2009).

• In the case of the ArbitraryProgrammingLanguage requisite, it can be found that PHP and Java are counted among the most popular programming languages (TIOBE 2010).

However, problematic in the context of generality is the above elucidated lack of reliability due to the limitation to a single software organization. For exploratory case studies – notwithstanding valid for the survey studies here too – Clutterbuck et al. (2009) remarked:

“[…] exploratory case study research results cannot be generalised beyond the studied SME and specific project. Normalisation will mitigate this limitation in as much as the results from this research may be compared within an overall context of agile method case study investigation.” – Clutterbuck et al. (2009, p. 18)

With normalization, it was meant to prove that it is about a “normal” case by considering measurable criteria. This is done here with a further characterization of HolidayCheck, inspired by criteria used by Hansson et al. (2006), Clutterbuck et al. (2009):

• There are five Scrum Teams, each constituted by 3 to 6 developers, 1 Scrum Master, and 1 Product Owner. The latter two are interchangeable members of a team.

• A sprint has a duration of 14 days.

• The size of a software development project is subdivided into three categories: All projects with less than 5 man-days (MD) of effort are treated as change requests, those with more than 37 MD as projects, and those in between as micro projects. These categories are distinguished by the degree of standardization in respect of requirements specification and project planning. The teams are engaged to 20% in change requests, to 40% in micro projects, and to 20% in projects. The remaining 20% are scheduled as a buffer for tasks that are not directly project-related.

• The lines of code, also known as LOC (Wikipedia 2010e), amounts about 2,000,000 regarding all maintained source code of the organization written with PHP or Java.

Evidently, the values above lie within the range of an usual application of Scrum or agile software development in general as there are no contradictions identifiable (see section 2.2). Finally, it can further be remarked that survey studies had been carried out at HolidayCheck before – internally as well as externally – in the form of employee attitude surveys. Therefore, the employees of this organization have already been prepared for taking part in a survey study.

5.2.2 Sampling

As evident from the above introduced research setting, the target population of the survey studies has been any SME company that develops web applications with any agile software development methodology so far. Now, the target population is further specified to be a software
development team within that company. While the former can be considered a macro-level sampling — in this case accomplished by handpicking a personally known company — the latter can be considered a micro-level sampling. For that micro-level sampling it is decisive to point out that the project stakeholders were designated to be over-represented due to the **HIGH-LEVEL BENEFIT** requisite as the object of research. Certainly, this requisite addresses rather project stakeholders than developers. To consider this fact, it was suggesting to overemphasize the role of the project stakeholders by letting attend more project stakeholders than developers. Therefore, the target population was definitely not a *typical* composition of a team regarding the proportion of the different professions like, for example, the 3-6/1/1 proportion of the regular Scrum Team at HolidayCheck (see section 5.2.1).

The micro-level sampling was accomplished by employing a non-probabilistic sampling technique followed by a probabilistic one: First, a self-selection sampling (Oates 2006, ch. 7) was done, where all developers, project managers, and product managers could express interest in taking part in the studies by responding to an appropriate email invitation addressed to them. Then, each request for participation was permitted by applying the first-come-first-served principle until the aimed proportion of professions was reached, what represents a form of stratified sampling (Oates 2006, ch. 7). The proportion was set relatively to the number of available participants: 40%/20%/40%. The rationale for this concrete proportion is given here:

- It was decided that, despite the desired over-representation of project stakeholders, the number of product managers must not exceed the number of developers. Doing so would alienate the initial target population too much, because the actual proportion of the overall available developers and product managers at HolidayCheck was 25 to 12 (see section 5.2.1). Anyway, by using the same relative number as for the developers the product managers are still considered with a distinct 108% over-representation compared with the number of developers.

- The small number of overall available project managers (4) implies that it would virtually be impossible to justify a relative number of project managers that equals or exceeds the relative number of developers or that of product managers. After all, the number of overall available developers is more than 6 times higher, and that of the overall available product managers still 3 times higher. On the other hand, the halved relative numbers of the others — developers and product managers — was seen as a minimum: A relative number of 20% requires 8 participants in total to make an absolute number of 1.6. That is the smallest number which can be interpreted discretely as 2 by rounding up. And a number of at least 2 participants sharing the same profession should be ensured to prevent biases of an otherwise sole participant of that profession. However, it was doubted whether more than 8 participants would take part at all. Therefore, the relative number of project managers was not further decreased.

Indeed, the resulting sampling size, i.e. the number of participants, was exactly 8. However, neither the participants in general nor the project managers revealed as the “bottleneck” in the sampling — that were the developers: The sample size could have been increased regarding the number of requests for participation from product managers and project managers. But no more than 3 developers requested a participation. This forced a 3/2/3 proportion — 3 developers, 2 project managers, and 3 product managers. Eventually, a response rate of 100% occurred in the subsequent study series as the participants took their attendance very seriously. Thus,
the participants kept their value throughout the study series such that the anyway susceptible reliability of the expected results – the sampling size met only the predetermined minimum number of participants – was not diminished further.

5.3 Research strategy

The evaluation of the techniques and tools by practitioners as an empirical answer to a large part of the research questions (see section 1.4) constitutes a largely non-exploratory research. For that, the employment of surveys is suggesting as that research strategy meets the aim of verification:

“Essentially, the researcher should have a very good idea of the answer before starting a survey. Thus, traditional survey research usually serves as a methodology of verification rather than discovery.” – (Gable 1994, p. 114)

On the other hand, finding answers to the exploratory question about a concrete benefit in terms of the HIGHLEVELBENEFIT requisite, i.e. how that could be met, is not necessarily neglected by employing surveys:

“Survey research is especially well-suited for answering questions about what, how much and how many, and to a greater extent than is commonly understood, questions about how and why.” – (Pinsonneault & Kraemer 1993, p. 6)

Compared to a case study, a survey study leads to results with a higher generalizability (Gable 1994), which was deemed important in the development of the research goals and the research questions in sections 1.3 and 1.4. Likewise confirmed by Gable (1994), a survey study comes with a higher repeatability. This could facilitate the comparability of the studies among each other. Thus, a verification of a technique relatively to another one would be made deducible.

Evidently, the exploratory portion of the evaluation demands the collection of qualitative data. For that, there are essentially two data collection methods suitable: Observations and interviews (Seaman 2002). However, the former is out of the question as many aspects of software development cannot be studied comprehensively by observations because “much of software development work takes place inside a person’s head” (Seaman 2002, sec. 2.1). And this kind of evaluation clearly is such a case, as only a productive employment of a technique would provide matters to observe. However, the actually decisive point why interviews were finally preferred to observations was of rather pragmatic nature:

“In a given setting, the resources available will be limited. A researcher would have to prioritize between, for example, carrying out several artificial versus few comprehensive, realistic experiments […]” – (Sjøberg et al. 2007, sec. 5.4)

In consideration of the fact that the resources – the number of potential participants as well as the time available for conducting the studies – were indeed limited, it was decided to follow the

21References to research methods for information systems are used here in addition to those for software engineering. This can be legitimated with the widespread but unjustified disregard of the former by software practitioners (Glass 2009).
latter option. And this implied to employ interviews instead of observations because interviews are obviously less people-intensive or time-consuming (Gable 1994, Sjøberg et al. 2007). In the context of the time consumption, it has to be pointed out that the idea of conducting a series of studies is not only effective but also efficient:

“It is desirable in interpretive studies to preserve a considerable degree of openness to the field data, and a willingness to modify initial assumptions and theories. This results in an iterative process of data collection and analysis, with initial theories being expanded, revised, or abandoned altogether.” – (Walsham 1995, p. 76)

Nonetheless, solely interviews for data collection were not seen as satisfying because a lack of structuredness of those was suspected. Without that, the quantitative data, necessary for the actual verification, could hardly be collected. Therefore, questionnaires were decided to be employed to ensure a largely structured interview. Even though the employment of questionnaires forms an own data collection method (Oates 2006, ch. 15), it can principally be combined with the employment of interviews. In the case of HolidayCheck, questionnaires were further eligible due to the familiarity of the employees with those (see section 5.2.1). Therefore, the questionnaires were determined to embrace the verbal interviews entirely such that the actual interviewing could occur in the form of questions from the participant themselves – arising by filling out the printed questionnaires. Moreover, the studies were planned to be conducted in one sole group without a dedicated interviewing of each single person. That again in order to ease the interviews, which are often perceived as an exam-like situation by the participants:

“It is also important to make it clear that there are no ‘right’ answers. Software developers sometimes mistakenly believe that anyone coming to interview them (or observe them) is really there to evaluate them.” – (Seaman 2002, sec. 2.2)

Finally, it can be deduced that the worked out research strategy is sound as “any software engineering issue is best investigated using a combination of qualitative and quantitative methods” (Seaman 2002, sec. 5). Nonetheless, much more interesting in the context of the thesis’ topic is the conformity of the research strategy with agile software development: While the study series itself coincides with the iteration concept of agile software development methodologies, the group interview resembles the team-based nature of those (see section 2.2). Hence, the research setting adhered to the productive setting of the software organization – where the evaluated techniques might actually be applicable.

5.4 Study series

Each of the four studies, except of the first one, addressed a single technique. Per week one study was performed. Apart from the questionnaire-based interview, a study comprised the presentation of those topics that are necessary to understand the outcome of the treated technique in form of a documentation artifact as well its context to the verification. Such a presentation was done verbally by means of presentation slides in English, as all of the participants have been proficient in English as a second language. An examples for such a slide can be seen in figure 8. The slides contained only little text in order to keep, besides the research strategy itself, also the implementation of that conform to the tenth agile principle about simplicity (see section 2.2.2). Likewise, it was striven for a minimization of the number of slides for each presentation.
Estimating
the effort and risk
of implementing a change request.
(Before planning and subsequent implementation!)

Figure 8: Example of a slide used in the presentation for the conduct of a study

Figure 9: Fully labeled Likert scale with five response categories as used in the questionnaires

Nonetheless, compared to the filling out of the questionnaires, the presentation constituted the larger part of a study, which had a duration of approximately one hour in total. The interview itself occurred through comprehension questions from the participants themselves, which they uttered during the presentation or while filling out the questionnaires. In both cases, to utter open questions was explicitly asked for.

Each presentation was subdivided into two parts. After each part, a questionnaire was filled out by the participants. Thus, there were two questionnaires per study. In the case of the first study, an additional questionnaire was to fill out before the presentation. All nine questionnaires can be found in appendix E. Each of them began with an introductory text and comprised mainly questions that (1.) were multiple-choice questions (2.) with a Likert scale (Likert 1932), which (3.) was fully labeled with five categories. Furthermore, the neutral category (4.) was placed at the end of the scale and (5.) labeled unambiguously (see figure 9). This was decided along the following considerations:

1. Multiple-choice questions are simply to answer (Oppenheim 1998), what again conforms to the actual setting due to the fulfillment of the tenth agile principle.
2. A Likert scale is particularly suitable and well-proven for querying attitudes (Oppenheim 1998, ch. 11).

3. A fully labeled Likert scale with five response categories is recommended according to the decision framework of Weijters et al. (2010) for designing Likert-scale questions. That framework considers the number of response categories and the labeling of those – here in the case of stand-alone studies for the purpose of an opinion measurement with a population from a non-academic setting.

4. Placing the neutral category at the end might possibly prevent an abundance of rather pointless mean responses. This problem is known as “midpoint responding” (Baumgartner & Steenkamp 2001) and was paraphrased by Tourangeau et al. (2004) as “middle means typical” in respect of the common placement of the neutral category in the middle of a Likert scale.

5. Furthermore, the neutral category can have the meaning of a “truly neutral position” as well as that of an “ambivalent position” (Nowlis et al. 2002). And in the context of a prevention of midpoint responding, the latter seems more desirable as neutrality is always easier to state than ambivalence. Therefore, the neutral category was labeled with “I am not sure.”

That Likert scale was also used – unchanged – for factual questions in order to maintain consistency, what increases the reliability of the results (Oppenheim 1998, ch. 8). Otherwise, free-text fields were used where required, or as a complement to the multiple-choice questions, which were designed close-ended by the Likert scale (Taylor-Powell & Marshall 1998). In the latter case, they offered a place to leave comments and, further, to annotate understanding difficulties. In general, concrete examples were added to most questions as recommended by Oppenheim (1998). Particularly because purely hypothetical questions should be avoided.

Subsequently in this thesis, each of the four studies will be treated with the following structure: First, the purpose of the study is abstracted. Then, the conduct of the study is portrayed with a focus on the presentation, including the number – written in square brackets – of the shown slides for each presented unit. This is followed by an elaboration of the design of the two (or three) questionnaires of the study. There, it is explained what exactly was queried, how it was queried, and why it was queried in that certain way. The questionnaires are treated one after the other. This separation can also be found in the subsequent illustration of the results obtained from the questionnaires and the interview, and in the final analysis of those.

5.5 Study 1

In the first study, the participants were prepared for the whole study series conducted for this thesis. For that, an introduction of the topic was given, accordantly with the introduction of the thesis in section 1 and completed with the very most relevant background knowledge from section 2. This was followed by a presentation of the research framework from section 3. Besides that necessary introduction of the participants into the topic in general, and the specific view on it through the research purpose (see section 1.3) and the research questions (see section 1.4), the first study was further supposed to deliver as an outcome a first evaluation of that view
by practitioners. This was set as a verification of the initial motivation and the worked out research framework. For proving the relevance of that verification, the participants were also surveyed on their professional experience and education.

5.5.1 Presentation

Before the presentation, the participants were asked to fill out questionnaire #1. The first part of the presentation comprised the introduction into the topic. The view on that, finally determined by the research purpose and the research questions, was explained mainly along the practice-oriented parts of the elaborated lack of research in program comprehension and agile software development as combined topics (see section 1.1) and the given rationale for supporting program comprehension by documentation (see section 1.2). However, the research questions themselves were not presented in detail as they were not particularly relevant for the study series. It was only necessary to briefly mention the research goals before proceeding with presenting the research framework – comprised in the second part of the presentation – as that is directly related to those. Concretely, the slides of the first part comprised:

1. Explanation of the topic with the help of a simplified version of the Venn diagram from figure 2 [1 slide]
2. Definitions of program comprehension and related terms from section 2.1 up to the notion of a mental model [1 slide]
3. List of the four agile values, with the second one highlighted as that is the most relevant in respect of the apparent contradiction between documentation and agile software development [1 slide]
4. Contrast of aspects of agile software development that support program comprehension and aspects of it that neglects program comprehension, likewise to the beginning of section 1 [1 slide]
5. Contrast of the understanding of documentation from agile software development and the aimed documentation devoted to program comprehension, likewise to section 1.2 [1 slide]
6. Explanation of the research goal of obtaining documentation techniques or tools that meet certain requirements [1 slide]

In the break, the participants were asked to fill out questionnaire #2. Subsequently, the presentation was continued with the second part, covering all requisites except the ARBITRARYAGILE-METHODOLOGY requisite and the ARBITRARYPROGRAMMINGLANGUAGE requisite as those are not of relevance for a software organization (see section 5.1). In the case of HolidayCheck, the predetermined agile software methodology is Scrum (or Scrumban), the programming languages PHP and Java (see section 5.2.1). Each of the remaining four requisites was presented in the following way:

1. Definition of the requisite in the form of a single sentence, accompanied by a more detailed verbal description of it
2. Outline of the rationale for the requisite along its derivation from the aim of supporting program comprehension and aspects of agile software development, likewise to the appropriate subsection in section 3

Agile software development was not deepened further because the participants have already been familiar with it, in contrast to program comprehension.
For each requisite 1 slide was used for presenting it, except for the HIGHLEVELBENEFIT requisite (2 slides) as for that it was inevitable to exemplify the two types of benefits (see section 3.2). Finally, the participants were asked to fill out questionnaire #3. For supporting that, the 5 slides about the requisites were printed as a handout for each participant.

5.5.2 Questionnaire design

**Questionnaire #1**  The first questionnaire comprised classification questions about demographic information – the professional experience and education of the respondent, i.e. the participant. This has been regarded crucial for relativizing the results of this study and, secondarily, those of the subsequent studies in the series to the participants’ level of professionalism. First, the position of the respondent – software developer, project manager, or product manager – was queried in question #1.1 for distinction in the analysis of the results according to the sampling (see section 5.2.2). The actual query of the professional experience was divided into three questions: the duration of the occupation at HolidayCheck (question #1.2), the duration of the overall work experience in the appropriate profession (question #1.3), and the duration of work experience in respect of agile software development in particular (question #1.4). Therefore, a later distinction between the familiarity with the appropriate profession in general, that with agile software development, and that with concrete processes in a particular software organization has been facilitated. As it could be assumed that the employees at HolidayCheck do not have much work experience due to their relatively young age (see section 5.2.1), and as agile software methodologies still can not be considered “traditional”, the scale for querying the experience in 1-year intervals was designed thus highly grained and appropriately limited. Finally, it was queried whether the respondent owns a university-level education that is relevant for the appropriate profession (question #1.5). This was included to further indicate the level of professionalism.

**Questionnaire #2**  The motivation for approaching to the topic as expounded in section 1 was verified with the questions in the second questionnaire. To prevent the respondents answering these questions of rather general nature too much narrowed to the software organization, it was pointed out in the introductory text of the questionnaire that the questions are not particularly related to HolidayCheck. Question #2.1 queried the respondent’s opinion on whether supporting program comprehension is particularly needed in the case of agile software development. Analogously, question #2.3 queried the respondent’s opinion on whether program comprehension can be supported by documentation without contradicting agile values and principles. In both questions the Likert scale was used. Even though the latter question was exemplified as usual (see section 5.4), only the second agile value and the tenth agile principle (see sections 2.2.1 and 2.2.2) were given as examples. It was not given a likewise concrete example for documentation as that has been the object of research. Because of the complexity of both questions #2.1 and #2.3, a comment field was added to each of them in the form of a trivial free-text question (questions #2.2 and #2.4).

**Questionnaire #3**  Contrary to questionnaire #2, the third questionnaire was explicitly related to the participants’ experience from the software organization by an appropriate introductory text. Due to the object of this questionnaire – the requisites from the research framework
– it would have been impractical to keep up a likewise generality as the requisites are linked to specific professions or, respectively, positions in a software organization. The questions followed a consistent pattern: First, it was queried the opinion on a certain issue for each of the four considered requisites with a series of Likert scales in a sole question. Then, a comment field for that question followed, again in the form of a free-text question. This pattern was repeated three times for querying the opinions on:

- Whether the requisite is justified (questions #3.1 and #3.2)
- Whether it is important for the respondent’s involvement in agile software development (questions #3.3 and #3.4)
- Whether it is important for the other stakeholders of the software organization (questions #3.5 and #3.6)

Actually, the question regarding a requisite’s justification was included to facilitate the respondent’s access to the question pattern and, further, a later validation of the answers to the more concrete questions #3.3 and #3.5: Related to these questions, question #3.1 is redundant to a certain degree because it is more or less implied that a respondent appraised a requisite as justified if this respondent appraised the requisite as important for either the own involvement or that of another stakeholder.

5.5.3 Results

**Questionnaire #1** Not surprisingly, the sampling was expectedly accomplished, as validated by the answers to question #1.1: three software developers, two project managers, and three product managers. Because they have already been represented as intervals, the answers to questions #1.2 and #1.3 about the work experience are depicted as histograms in figures 10a and 10b here. The median experience can be estimated to 3 years for the former and to 5 years for the latter. Contrary to that, the experience with agile software development (question #1.4) is considerably shorter. Except for one participant, who stated to had 1-2 years experience, all participants had less than 1 year experience with agile software development. The answers to question #1.5 about the education are depicted in the form of a pie chart in figure 11. In this context, it has to be mentioned that all participants who stated to have no related education have been either product managers or project managers. Furthermore, all of those stated additionally – verbally – to own a university degree as they considered the phrasing of the question as hard to understand. Besides that, some of the participants considered the phrasing of the questions in general as too long.

**Questionnaire #2** The answers to the Likert-scale questions #2.1 and #2.3 are shown in the form of stacked bar charts in figures 12a and 12b. This kind of chart will be used for depicting answers to Likert-scale questions from now on, as it is most suitable for illustrating relative proportions as well as absolute frequencies. Recognizing the proportions is facilitated here by an appropriate coloring of the stacks, which represent the different response categories from left to right beginning with “no”. Recognizing frequencies is likewise facilitated by the horizontal axis, which is discretely scaled according to the maximum possible number of answers. In this context, the answer “I am not sure” is not seen as an answer to be considered in the stacked bar
Experience at HolidayCheck (question #1.2)

Experience at all (question #1.3)

Figure 10: Histograms of the participants’ work experience as a software developer, project manager, or product manager

Figure 11: Pie chart of the participants’ education that is directly related to their positions as a software developer, project manager, or product manager (question #1.5)

This is more conform to the original intention behind the positioning and labeling of that category in the questionnaire (see section 5.4). Additionally to the stacked bar charts in this section, the answers to all Likert-scale questions are listed as raw data in appendix F.

The free-text questions #2.2 and #2.4 were filled out with comments by four participants: SD1, SD2, SD3, and PJM1 (see appendix F). Most conclusive were the following comments:

- SD1 substantiated a “yes” answer to question #2.1 with the claim that a program is probably more durable than the formation of a developer team which maintains it.
- SD2 commented a “rather yes” answer to question #2.1 with the assumption that “scattered code units” are more typical of agile software development.
- A “rather no” answer to question #2.1 by PJM1 was justified with the assumption that due to the cross-functional teams in Scrum (see section 2.2.5), any increased need for supporting program comprehension would be eliminated.
- SD1 mentioned again a long-term aspect appendant to a “rather yes” answer to question #2.3 by claiming that representations of data in the code are not changed often – despite

All software developers among the participants have been coded with the prefix “SD”, the project managers with “PJM”, and the product managers – not appearing here – with “PDM.”

(a) “Do you think that agile software development generally comes with an increased need for program comprehension [...]?” (question #2.1)

(b) “Do you think that program comprehension can be supported with documentation such that it does not contradict agile values and principles?” (question #2.3)

Figure 12: Stacked bar charts in respect of the thesis’ topic in general (questionnaire #2)

agile software development.

- A “rather yes” answer to question #2.3 by SD3 was justified with the annotation that the additional effort of documenting remains critical.
- A “rather yes” answer to question #2.3 by PJM1 was commented with the suggestion of enabling this kind of documentation by explicitly considering it in the “definition of done” (Schwaber & Sutherland 2010) in Scrum.

Questionnaire #3 Likewise to questions #2.1 and #2.3, the Likert-scale questions #3.1, #3.3, and #3.5 are depicted here with stacked bar charts in figures 13a, 13b, and 13c. Here, the relevant comments from those that were given as answers to the complementary free-text questions #3.2, #3.4, and #3.6 are listed:

- PJM1 added to a “yes” answer to question #3.1b (the HIGHLEVELBENEFIT requisite) that a comprehensibility for all stakeholders of such an extracted high-level documentation was of particular importance.
- SD1 justified a “rather yes” answer to question #3.1c (the NOSEPARATEARTIFACT requisite) in place of a “yes” answer with the inevitableness of producing separate documentation for developers and for project stakeholders anyway.
- PJM1 explained a “no” answer to question #3.3c (the NOSEPARATEARTIFACT requisite) with being comfortable with explicit requests for documentation – also by developers – in the software organization.

A validation of the answers to questions #3.3 and #3.5 reveals that the results are sound: If a participant answered in a certain tendency to question #3.1 this could mostly be found as well either in an answer to question #3.3 or in an answer to question #3.5. There were only two distinct occurrences of a contrariness – a mere “rather yes” answer to question #3.1a compared to two “yes” answers to questions #3.3a and #3.5a, and an exceeding “yes” answer to question #3.1a compared to a “rather yes” answer to question #3.3a and a “rather no” answer to question #3.5a.

All developers among the participants affirmed the importance of the LOWLEVELCONTEXT requisite for themselves with a “yes” answer. Contrary to that, the project stakeholders affirmed
(a) “Do you think that the following requisite is justified in respect of a conceivable application of it in the agile software development at HolidayCheck?” (question #3.1)

(b) “Do you think that the following requisite is of importance for your involvement in the agile software development at HolidayCheck?” (question #3.3)

(c) “Do you think that the following requisite is of importance for the involvement of other stakeholders in the agile software development at HolidayCheck?” (question #3.5)

Figure 13: Stacked bar charts in respect of the requisites (questionnaire #3)
likewise the importance of the HighLevelBenefit requisite for themselves. Regarding the answers of particular participants, those of SD3 are remarkable as the entire questionnaire #3 was answered with “i am not sure” by that participant. On inquiry, SD3 explained that the presentation of the requisites had not been comprehensible enough to state any well-grounded answer. A subsequent discussion revealed that the other participants had no such severe problems to understand the requisites. However, SD2 and PDM2 appraised the principle of naming the requisites as well as the concrete naming itself as irritating.

5.5.4 Analysis

Questionnaire #1  Not all of the participants had extensive work experience at Holiday-Check. However, according to figure 10b, they had at least 3 years overall work experience. Therefore, they can evidently be regarded as professionals instead of as recent graduates. The fact that a few of the participants obviously were relatively new to the software organization can be neglected: Due to an enormous increase of employees, software development processes as well as processes in general have been changed frequently in this software organization (see section 5.2.1). Actually, this might have lead to an increased generality of the results of the study series and thus to a further generality of the thesis. Contrary to that, the conclusiveness of deductions assuming extensive experience with agile software development is per se limited because of its novelty to the participants. The fact that a large part of the participants owns a university degree substantiate their professionalism – even if the degree is not directly related to their profession like in the case of the project managers and the product managers. This can be justified with the lack of appropriate university programs that are as specifically connected to practice – in contrast to computer science or software engineering programs in the case of software development (Pant & Baroudi 2008, Downey 2010).

Apparently, the phrasing of the questions has not been optimal. However, through additional verbal explanations unclarity among the participants could be eliminated to a large extent. Furthermore, this finding was considered in the questionnaire design of the subsequent studies.

Questionnaire #2  Even though there are more “yes” and “rather yes” answers than “no” and “rather no” answers to question #2.1, the affirmation of it by the participants is not as distinct as in the case of question #2.3 (see figures 12a and 12b). A possible explanation for this might be that the issue of a potentially increased need for supporting program comprehension in agile software development was addressed in the presentation along the elaboration of that from section 1.1. That section is much more focused on theories than section 1.2, on which the presentation of the rationale for supporting program comprehension by documentation for question #2.3 based. Probably, for the participants – practitioners after all – the latter was simply more understandable. Anyway, it can be deduced that the need for supporting program comprehension was not seen as decreased by them either. In fact, the answers to free-text question #2.2 by SD1 and SD2 are practice-oriented arguments for an increase.

On the other hand, the distinct affirmation in the case of question #2.3 has to be relativized as it is doubtful whether the participants have fully understood the question: While question #2.1 comprised only two topics – program comprehension and agile software development –
question #2.3 comprised additionally a third topic – documentation. Obviously, the interrelations between these three topics are more complex. The comments given as answers to free-text questions #2.4 at least did not disprove this speculation as none of them address all three topics explicitly.

**Questionnaire #3**  The presented requisites seem to be basically understood by the participants. At least in the case of the LOWLEVELCONTEXT requisite and the HIGHLEVELBENEFIT requisite, this is substantiated by the “preference” of the former by the developers and that of the latter by the project managers and product managers. It would be surprising if the importance of these two requisites were not actually balanced in this tendency, because the LOWLEVELCONTEXT requisite is obviously more relevant for a developer, as well as the HIGHLEVELBENEFIT requisite is more relevant for a project stakeholder. The growing unsureness toward the end in the case of questions #3.3 and #3.5 (see figures 13b and 13c), i.e. more “i am not sure” answers, indicate that the NOSEPARATEARTIFACT requisite and the PRIMARILYPRIORITIZEDCOMPRESSED requisite were either not understandable or not appraisable for the participants. The fact that there is no likewise growth of unsureness in the case of question #3.1 (see figure 13a) and that the participants’ comments embody thoughtful reflections also on the NOSEPARATEARTIFACT requisite speaks for the latter. In this context, it has to be pointed out that the general unsureness in the case of question #3.5, compared with question #3.3, is not extraordinary as the question was unrelated to the own involvement. Overall, the results of questionnaire #3 can be taken as an affirmation of the research framework from section 3, even though the participants had difficulties with the representation of that in the form of the requisites. That might have been the case due to the scientific grounding of it necessary for the thesis. Whether that necessity applies also inevitably to a presentation of it for practitioners remains an open question.

### 5.6 Study 2

The second study embodied the evaluation of the fundamental technique of the tool of Cornelissen et al. (2007) as summarized and discussed in section 4.6 – generating scenario diagrams from unit tests. As the evaluation was limited to the HIGHLEVELBENEFIT requisite (see section 5.1), the study focused on that as an object of research. However, that requisite as well as the other ones were not addressed explicitly: The question has been whether there is a direct benefit of the resulting graphical documentation for a project stakeholder and, further, what benefit that is. An examination of such a benefit in respect of more detailed aspects of the HIGHLEVELBENEFIT requisite, like an identification of the benefit’s definite type or the exact procedure how to extract it, was not determined as essential for proving the meeting of the HIGHLEVELBENEFIT requisite. Thus, an inclusion of that requisite or another one in its original representation would only have distracted the participants from the essential questions – whether and what. This is substantiated by the participants’ appraisal of the requisites as “overformulated” in the first study (see section 5.5.3). Further in this context, the execution of the actual technique was not comprised in the study but only the results of that – a certain kind of sequence diagrams. Certainly, these diagrams were presented as well as the fundamentals of a change impact analysis, as the hypothesis has been that the benefit in the case of this technique is mainly constituted by a helpfulness for that.
As the first study revealed that some of the participants had difficulties to comprehend the presentation as well as the questionnaires (see section 5.5.3), it was decided to focus more on comprehensibility in the study series from now on: First, it was striven for shorter questions in the design of the questionnaires. Second, repetitions of issues dealt with in the preceding study – here those of the first one – which were also relevant for the recent study were included in the presentation. Additionally, appropriate questions were added in the questionnaires for a verification of comprehension, whereas free-text questions were withdrawn from the questionnaires where not actually needed. In place of them, a face-to-face conversation was asked for, which was supposed to eliminate any unclarity better than a field for annotations. Besides, this makes the conduct of a study even more conform to agile principles – as desired (see section 5.4).

5.6.1 Presentation

In the first part of the presentation, it was dealt with sequence diagrams as that was necessary for querying – via questionnaire #4 in the subsequent break – the understandability of those among the participants. Such an understandability is obviously most crucial for deducing a benefit of the diagrams for a project stakeholder. In this context, it was not explicitly distinguished between sequence diagrams and scenario diagrams (see section 4.6). Even though questionnaire #4 contained scenario diagrams, these were not referred as such. And the presentation about sequence diagrams was not such deep that any issue was addressed which differs from the understanding of sequence diagrams as scenario diagrams. In detail, this presentation comprised the following issues:

1. Repetition of the topic with the simplified version of the Venn diagram from the first study (see section 5.5.1) for strengthening the comprehension of it among the participants [1 slide]
2. Characterization of a sequence diagram as an interaction diagram, likewise to its introduction in section 4.6.1 [1 slide]
3. Explanation of the component parts of a sequence diagram by means of exemplary diagrams from Kimmel (2005, ch. 4) [3 slides]
4. Demonstration of a sequence diagram that depicts a concrete scenario completely (see figure 14), likewise taken from Kimmel (2005, ch. 4) [1 slide]

The second part of the presentation dealt with the concept of a change impact analysis. Thus, the participants could grasp sufficient knowledge for subsequently filling out questionnaire #5 about the potential helpfulness of the presented sequence diagrams for a change impact analysis. Furthermore, it was queried by this questionnaire if there could be another direct benefit apart from that, as the study only focused on a facilitation of a change impact analysis but was not limited to that. The second part of the presentation comprised in detail:

1. Explanation of a change impact analysis in general, likewise to its first mention in section 4.6.2 [1 slide]24
2. Contrast of traceability analysis and dependency analysis according to Arnold (1996, pp. 1-2) [1 slide]

24see figure 8
3. Reference to the derivation of the sequence diagrams in questionnaire #4 – generated from unit tests – to clarify the context to a change impact analysis, likewise to the discussion in section 4.6.2 [1 slide]

5.6.2 Questionnaire design

**Questionnaire #4**  The questionnaire #4 about sequence diagrams began with a comprehension question whether the respondent has grasped the concept of sequence diagrams in principle (question #4.1). Question #4.2 contained the presented exemplary diagram (see figure 14) for querying the understandability of sequence diagrams by means of a concrete, as-simple-as-possible example – complementary to the purely textual question #4.1. This diagram could be considered as simple due to its explanatory nature and its use for the exemplification of sequence diagrams in the presentation before. Furthermore, the depicted login scenario could be taken as granted as it can typically be found in the web applications developed at HolidayCheck (see section 5.2.1). In the questionnaire, understanding was defined as achieved if the respondent could interpret the objects and messages depicted in the diagram. Questions #4.3 and #4.4 were intended for an investigation of the understandability of scenario diagrams compared with that of sequence diagrams, i.e. of the question whether generated sequence diagrams are less understandable. This was done by two questions because it was further distinguished between a semi-automatically generated diagram (question #4.3) and a fully automatically generated one (question #4.4). In section 4.6.2, it was found as doubtful whether the latter is useful despite its complexity. Likewise to question #4.2, concrete example diagrams were contained in both questions – the diagram from figure 5 in question #4.3 and that from figure 22 in question #4.4. As the latter diagram is the unabstractions version of the former one (see section 4.6), question...
#4.4 was formulated in relation to question #4.3, i.e. the respondent was asked whether the interpreted objects and messages from that question could be recognized again. The formulation of question #4.3 itself was analogous to that of question #4.2. It was only added that the diagram depicts a certain run of a Pac-Man implementation.

**Questionnaire #5** Like question #4.1, the first question of questionnaire #5 has been a comprehension question – now regarding the concept of a change impact analysis. The rest of this questionnaire referred to questionnaire #4 such that no further example diagrams had to be introduced. In question #5.2, the respondent was asked whether the sequence diagram contained in question #4.2 could be helpful for a change impact analysis. For exemplification, a contextually related change request – adding a “change password” function – was included in the question. Like in the context of understandability (questions #4.3 and #4.4), it has been unclear whether a generated sequence diagram is as useful as a non-generated one – now in the context of a change impact analysis. Therefore, question #5.3 queried, in comparison with question #5.2, the helpfulness of the diagram contained in question #4.4 for a change impact analysis. Contrary to questionnaire #4, it was not distinguished between the semi-automatically generated diagram and the fully automatically generated one this time as the differences were supposed to be deducible from the results of questions #4.3 and #4.4: If the abstracted scenario diagram – generated semi-automatically – revealed as more understandable than the unabstracted one – generated fully automatically – the former would be at least as helpful for a change impact analysis as the latter. As a familiarity with the object of research can be assumed after answering the questions #5.1 to #5.3, the general helpfulness of sequence diagrams for a change impact analysis was investigated more detailed – apart from any concrete diagram – in question #5.4. There it was distinguished between a helpfulness for estimating the effort for an implementation of the change request (question #5.4a) and that for estimating the involved risk potential (question #5.4b). A later validation of the answers to questions #5.2 or #5.3 was not planned to be facilitated by question #5.4 like in the case of question #3.1. Finally, free-text question #5.5 asked the respondent, as mentioned above in section 5.6.1, whether another direct benefit of the sequence diagrams is imaginable.

### 5.6.3 Results

**Questionnaire #4** Question #4.1 was answered by the participants with 6 “yes” answers and 2 “rather yes” answers. Like in section 1.2, the other Likert-scale question of this questionnaire – questions #4.2 to #4.4 – are shown in the form of stacked bar charts in figures 15a, 15b, and 15c. Through conversation it was revealed that not all participants had known the video game Pac-Man by its name. However, as all participants knew the basic principles of that game and, of course, those of video games in general, this unclarity could be eliminated.

**Questionnaire #5** Except for question #5.1, the answers to all Likert-scale questions – questions #5.2 to #5.4 – are depicted in figures 16a, 16b, and 16c. Likewise to the answers to question #4.1, those to question #5.1 are mentioned only textually here: 3 “yes” answers and 5 “rather yes” answers. Regarding the answers of particular participants, those of SD1 clearly differ from those of the other participants: SD1 answered “no” to question #5.2 and “rather no”
to question #5.3, whereas the other participants answered question #5.2 compared with question #5.3 always more or equally affirmatively.

Question #5.5 was answered by all participants except PDM2. However, most of these answers actually restated more or less explicitly the facilitation of a change impact analysis as an “additional” benefit, or were constituted by comments on that. For example, PJM1 pointed out that, with an appropriate training course, a project stakeholder could probably be enabled to use sequence diagrams in a helpful manner for a change impact analysis. As an actually additional benefit, SD3 stated that semantic relations in the code became clear by such sequence diagrams. Thereby, the support of program comprehension was confirmed. However, that was not anymore an object of research. Similarly in this context, SD2 and PDM3 confirmed that the diagrams could constitute a form of documentation – again, not anymore an object of research. In fact, the only answer which owned sufficient novelty to meet the question was that of SD1, who stated that the sequence diagrams could be considered as an indicator of source code quality: the more complex a diagram, the worse the quality of the code from which it was generated.
5.6.4 Analysis

Questionnaire #4  From the answers to questions #4.1 and #4.2 it is deducible that the participants have understood the concept of a sequence diagram. Even though the diagram in figure 5 was an automatically generated one, it reveals – through a comparison of the answers to questions #4.2 and #4.3 (see figures 15a and 15b) – as not less understandable as the diagram in figure 14. On the one hand, you can argue that question #4.3 was not as strictly formulated as question #4.2, because it was asked whether any essential part of the diagram can be understood (see appendix E). On the other hand, you can oppose that the diagram of question #4.3, contrary to that of question #4.2, was not explained at all in the presentation. A slight evidence for a reduced understandability can be found for the fully automatically generated diagram (see figure 15c) in the case of question #4.4, as two participants answered “rather no” to it.

Questionnaire #5  Apparently, the concept of a change impact analysis was principally understood as all participants answered question #5.1 affirmatively, i.e. either with a “yes” answer or with a “rather yes” answer. However, the answers are, overall, not as affirmative as those...
to question #4.1 in the preceding questionnaire – only 3 “yes” answers in contrast to 6 “yes” answers (see section 5.6.3). Likewise, the answers to question #5.2 are less affirmative than those to question #4.2, of which question #5.2 was not only an equivalent in terms of querying understandability by means of a concrete example but it also referred to the diagram contained in the question #4.2 for that. Nonetheless, this “educational” example diagram is confirmed as helpful for a change impact analysis as all answers to question #5.2 except for one are affirmative. Contrary to that, the “realistic” example diagram, i.e. the semi-automatically generated one, seems to be rather not helpful because the non-affirmative answers to question #5.3 (see figure 16b) outweigh the affirmative ones. Due to the finding of a reduced understandability of the fully automatically generated diagram from the analysis of questionnaire #4 above, it is to assume that such a diagram is even less helpful than a semi-automatically generated one. Still, there can be found a certain degree of helpfulness of generated sequence diagrams for a change impact analysis: First, at least SD1 appraised the (fully automatically) generated diagram not as less helpful than the exemplary one. Actually, that participant appraised the former as more helpful than the latter. Second, the comment by PJM1 from question #5.5 indicates that the helpfulness of the diagrams – for whatever – is rather a matter of training. And in any case, a minimal helpfulness in respect of estimating the effort can be deduced doubtlessly from the stacked bar chart in figure 16c.

Obviously, the answers to question #5.5 do largely not lead to a further benefit in terms of the HIGHLEVELBENEFIT requisite. If SD2 and PDM3 had not only addressed the documentation aspect but also the abstraction level of the formed documentation or the usefulness of that, perhaps such a benefit would have been discernible. However, at least the idea of SD1 can be regarded as a basis for a benefit of the second type (see section 3.2). At the first sight, the stated relation between complexity and code quality might be contradicting: First, a sequence diagram generated fully automatically by the tool of Cornelissen et al. (2007) depicts simply more information (see section 4.6). Without any doubt, this does not indicate a bad code quality. Second, the relation seems to violate common object-oriented design principles such as, for example, “favor object composition over class inheritance” (Gamma et al. 1995, p. 20) or “strive for loosely coupled designs between objects that interact” (Freeman et al. 2004, p. 53). Complying with these principles would certainly result in more messages between objects. Thus, if complexity is defined here as the number of objects or of messages, this idea is useless for a practical application. However, if complexity is defined in a way that is conform to the mentioned design principles and that is conceivably recognizable as a pattern in sequence diagrams, then this idea could indeed lead to an indicator of a bad code quality.

5.7 Study 3

In the third study, the fundamental technique of the tool of Ratanotayanon et al. (2009) was evaluated (see section 4.7). Likewise to the preceding study, neither the HIGHLEVELBENEFIT requisite nor the application of the technique itself was addressed explicitly, but the result of the latter – visualizations of links from source code to user stories. From these visualizations only those were considered which do not require non-trivial programming knowledge to be understood. Otherwise, it would have been pointless to investigate the usefulness of those for the project stakeholders among the participants.
5.7.1 Presentation

In the first part of the presentation it was dealt with the linking concept as well as the visualizations of that. Analogously to questionnaire #4 from the second study, the understandability of both among the participants was queried via questionnaire #6. This part of the presentation comprised the following issues:

1. Explanation of the linking concept and the involved artifacts, likewise to the beginning of section 4.7.1 [1 slide]
2. Categorization of the possible abstraction levels of a link – code file, code line, and code element – aligned with the linking possibilities of the tool of Ratanotayanon et al. (2009), likewise to section 4.7.1 [1 slide]
3. Demonstration of three relevant visualizations of links regarding the presented abstraction levels (see figures 6a, 6b, and 17), taken from Ratanotayanon et al. (2009, p. 29) [1 slide]

Not only for the latter issue illustrations were used but also for the presentations of the abstraction levels. For that, the code of a “Hello World” program was displayed as that is typically the most simple program written with a specific programming language – in this case Java (Wikipedia 2010c). Thus, the concept of a code file, a code line, and a code element could be explained in a way most probably understandable for project stakeholders. It was decided to show, besides the overview with “blocks” and “stripes” (see figure 6a), and the hierarchical list of code elements (see figure 6b), also the modified explorer view of Eclipse (see figure 17) as a third visualization to map all three presented abstraction levels most comprehensively. Names for these visualizations were not stated at all, neither in the presentation nor in the questionnaire, to not confuse the participants unnecessarily.

The second part of the presentation was, again, about a change impact analysis. Contrary to the second study, the derivation of the resulting documentation artifacts, i.e., the visualizations, had not to be mentioned separately as the feasible application of the technique is obviously much easier to understand in this case. Apart from that, a repetition of the fundamentals of a change impact analysis was given with the explanation of a change impact analysis in general and the contrast of traceability analysis and dependency analysis from the first study (see section 5.6.1). Finally, the participants filled out questionnaire #7, which, analogously to questionnaire #5, queried the potential helpfulness of the artifacts from the considered technique for a change impact analysis.
5.7.2 Questionnaire design

**Questionnaire #6**  Analogously to question #4.1, question #6.1 queried whether the respondent has principally grasped the concept of the technique or the resulting documentation artifacts, respectively. In question #6.2, this was further queried with a concrete example of a visualization of the links, likewise to question #4.2 regarding understandability. To keep, as in the preceding studies, the questionnaire consistent with the presentation, the presented visualizations were used as three instances of such an example. Moreover, querying whether the respondents recognize the abstraction levels in the visualizations could be investigated in detail by those. This was considered as the finally achieved understanding of the linking concept. Contrary to question #4.2, the formulation of question #6.2 was stricter as the expression “for the most part” was not used. Using that would not have been adequate due to considerably less complexity: While in a sequence diagram several kinds of relationships – the different types of messages – are depicted, in a visualization of links from source code to user stories only one unidirectional kind of relationship is depicted. Thus, it can be taken for granted that a visualization is understood either entirely or not at all.

**Questionnaire #7**  The first question of questionnaire #7 was not, like question #5.1, a comprehension question, because a change impact analysis was treated only as repetition this time. Instead, question #7.1 was, analogously to question #5.2, about the helpfulness of the visualizations from the preceding questionnaire for a change impact analysis – again queried separately for each of the three visualizations. Like in question #5.2, a concrete example for a change request contextually related to the documentation artifacts was given. That was to consider Blu-ray Discs as data storage media in addition to DVDs, because it is obvious that the visualizations taken from Ratanotayanon et al. (2009) depicts a DVD vendor system or something similar (see figures 6a and 6b). Questions #7.2 and #7.3 were completely identical with questions #5.4 and #5.5, apart from the fact that they are related to the visualizations instead of to the sequence diagrams.

5.7.3 Results

**Questionnaire #6**  Question #6.1 was answered by the participants with 7 “yes” answers and 1 “rather yes” answer. The answers to question #6.2 are shown as a stacked bar chart in figure 18.

**Questionnaire #7**  The answers to the Likert-scale questions #7.1 and #7.2 are depicted in figures 19a and 19b. The free-text question #7.3 was answered by SD1, SD2, SD3, PJM1, and PDM2. Among them, SD3, PJM1 and PDM2 only substantiated their affirmative answers to questions #7.1 and #7.2 by appropriate comments or pointed out the documentation aspect of the visualizations. Like in the case of the answers to question #5.5 (see section 5.6.3), there was, again, only one answer which owned sufficient novelty to meet the actual question: SD3 proposed a kind of code coverage for user stories analogous to the code coverage known from unit tests (see section 4.5): An incomplete implementation of a user story could be identified by a missing link to it.
5.7.4 Analysis

**Questionnaire #6**  From the answers to question #6.1, it is obvious that the concept of links from source code to user stories inclusive the visualization of those has been understood by the participants. However, the understandability of the example visualizations (question #6.2) seems to be not as high as that of the example sequence diagrams from questionnaire #4 (see section 5.6.3). This might be grounded rather in the understanding of the linking concept and not in the understanding of possible visualizations because all three visualizations, according to the answers to questions #6.2a to #6.2c, were understood approximately equally well.

**Questionnaire #7**  As the answers to questions #7.1 are predominantly affirmative – more affirmative than those to analogous question #5.2 – it can be deduced that visualizations of links from source code to user stories were considered by the participants at least equally helpful for a change impact analysis. Like in the second study (see section 5.6.3), the understandability of a documentation artifact is higher than the helpfulness of it for a change impact analysis. However, that is a relatively trivial finding, as the former is a prerequisite of the latter. More remarkable is the fact that question #7.2a – about estimating the effort – received less affirmative answers than question #7.2b – about estimating the risk potential – whereas these proportions were inverted in the case of analogous questions #5.4a and #5.4b. This might be an evidence that the one technique is actually more helpful for estimating the effort and the other more helpful for estimating the risk potential. Unfortunately, the answers to free-text question #7.3 do not reveal any further appraisal that could underpin that speculation.

The proposal by SD3, given as an answer to question #7.3, could feasibly result in a further benefit in terms of the HIGHLLEVELBENEFIT requisite. Likewise to the code coverage in respect of the unit tests, a code coverage in respect of user stories could be a size relevant for quality assurance. A higher coverage in respect of the unit tests is virtually in any case desirable in the context of this thesis: First, it complies with agile software development, despite the costs of an increase of the coverage (Ellims et al. 2004). Second, as discussed in section 4.5.2, an increase of the coverage can come with an increase of the support of program comprehension. This is also conceivable for the coverage in respect of the user stories. The most suggesting course how the idea of such a size could actually evolve to a direct benefit is its incorporation into the
concept of acceptance tests, because these are, like unit tests, covered by the testing practice of agile software development. But contrary to unit tests, acceptance tests commonly address user stories explicitly (see section 2.2.3). Consequently, a code coverage in respect of tests and one in respect of user stories could be identical sizes in principle.

5.8 Study 4

Finally, the technique of producing documentation based on DR was evaluated, similarly to the preceding two studies. However, contrary to those, no particular tool or research work was considered this time. It was focused on the representation of a DR as a documentation artifact resulting from a documentation process that comprises such a technique. Furthermore, the evaluation was limited to informal representations. After all, more formal representations were discussed as an impediment to agile principles in section 4.8.2. Again, the questionnaires were designed analogously to those of the preceding study to query the understandability of informally represented DR as well as their potential helpfulness for a change impact analysis and beyond.

5.8.1 Presentation

DR and their representation was dealt with in the presentation by addressing the following issues:

1. Definitions of a DR and related terms from the beginning of section 4.8.1 [1 slide]
2. Categorization of DR representations regarding how formal those are, likewise to 4.8.1, including mention that only informal representations are relevant [1 slide]

3. Demonstration of two self-developed informally represented DR (see quotations below) [2 slides]

Any demonstrated example of a DR had to be suitable for the research setting made up by the participants. Thus, it had to be related to the implementation of a part of a program in order to support program comprehension, as well as it had to conform to agile software development. Furthermore, it was desirable that it does not lie outside the domain of the participants, to avoid further explanations. After all, every example used in a presentation for a study had to be as simple as possible in general. None of these three requirements have been fulfilled by the DR example from figure 7b discussed in section 4.8. The other DR example, from figure 7a, might have been simple enough for the participants. However, it would have led to a distinct researcher bias (Kitchenham & Pfleeger 2008): It refers literally to risk, while in the later question #9.2 the estimation of a risk potential is the subject. Certainly, the answers to that question would have been influenced in an affirmative direction. As through further research no suitable examples could be found, it was decided to develop fictitious ones in a purely textual form. Developing more than one example was suggesting as in the preceding studies always more than one example was employed. Developing exactly two examples was even more suggesting as the one could conform more to first definition of a DR and the other one more to the second definition (see section 4.8.1). By means of practical experience (see section 1.7) they were eventually developed, inspired by the login scenario from the second study (see section 5.6.2), which the participants have already been familiar with:

« The user name is not case-sensitive. However, in the authentification process case sensitivity is considered for the error message in the case of a failed login due to an incorrect user name. In this message the incorrect user name is re-stated. If that were lowercased by the system in order to process it further, the user could be irritated because the user name might have been entered case-sensitively. Therefore the user name is represented here by two variables. One holds the user name as originally entered, the other one as used internally. »

« In the authentification process the password is passed together with the date when it has been set such that it can be determined to encrypt it either with MD5 or with SHA-2. This is necessary for enabling the matching with the hash stored in the database. Passwords that have been set after 2008-10-01 were stored in the database encrypted with SHA-2 instead of with MD5, because MD5 is not considered secure anymore. As an alternative to that the obsolete MD5 hashes could have been replaced by SHA-2 hashes. But doing so would have required to re-set the affected passwords because they cannot be retrieved from their hashes. »

Actually, as there was no second repetition of the fundamentals of a change impact analysis – the last and only one was in the third study (see section 5.7) – there was no second part of the presentation. Consequently, both questionnaires #8 and #9 were filled out directly one after the other.
5.8.2 Questionnaire design

**Questionnaire #8** Analogously to questions #4.1 and #6.1, question #8.1 queried whether the respondent has principally grasped the concept of a DR. Question #8.2, the analog to questions #4.2 and #6.2, comprised, of course, the two presented DR. The understanding of the concept of a DR was defined as achieved when the respondent could recognize the decision as well as its rationale (see section 4.8). Again, the expression “for the most part” was not used for the same reason discussed in section 5.7.2.

**Questionnaire #9** Questionnaire #9 was largely identical with questionnaire #5 and even more identical with questionnaire #7, at which question #9.1 was the analog to question #7.1, question #9.2 that to question #7.2, etc. Essentially, only the referred technique and documentation artifacts differed. The development of the contextually related change request for question #9.2 was critical, because any more or less direct relation to the decisions stated within the DR might have led to another potential occurrence of a researcher bias. Hence, the development of the contextually related change request for question #9.2 was not only inspired by the login scenario from the second study, it was exactly the same as used in question #5.2: Adding a “change password” function.

5.8.3 Results

**Questionnaire #8** To question #8.1 4 “yes” answers and 4 “rather yes” answers were given by the participants. The answers to question #8.2 are shown as a stacked bar chart in figure 20.

![Stacked bar chart](image)

**Figure 20**: Understandability of the two examples of a design rationale (question #8.2)

**Questionnaire #9** Figures 21a and 21b depict the answers to the Likert-scale questions #9.1 and #9.2. The free-text question #9.3 was answered by SD1, SD2, PJM1, PJM2 and PDM2. Again, most of them – SD2, PJM1, PJM2, and PDM2 – substantiated affirmative answers to questions #9.1 and #9.2 by appropriate comments. Interestingly, PJM1 mentioned implicitly that DR could obviate reverse engineering in the case of a later change. Certainly, that would diminish the need for program comprehension. Much more explicitly, SD1 referred to the scenario of an “unrefactorable” obfuscating code from the introduction (see section 1.2). In this context, a suggestion by SD1 was to prevent any futile refactoring attempt by other developers through a DR that justifies an “unclean” part of the code.
5.8.4 Analysis

Questionnaire #8  Clearly, the answers to question #8.1 are less affirmative than those to the analogous questions #4.1 and #6.1 (see sections 5.6.3 and 5.7.3). However, it would be over-hastily to deduce that the concept of DR was not as well understood as the concepts of the other techniques. After all, every single answer to question #8.1 is affirmative. And at least the examples in this study were understood better than the examples in the other two studies when considering the highly affirmative answers to question #8.2, compared with those to questions #4.2 and #6.2. Question #8.2a – about the first quoted DR above (see section 5.8.1) – received slightly more affirmative answers than question #8.2b – about the second quoted DR. However, the difference is marginal. Thus, nothing non-trivial can be deduced from that.

Questionnaire #9  The answers to all questions of questionnaire #9 are even more affirmative than those to the questions of questionnaires #5 and #7 (see sections 5.6.3 and 5.7.3). Therefore, it can be deduced that a DR as a documentation artifact has been appraised as highly helpful for a change impact analysis by the participants. However, the generality of this deduction is questionable because the treated examples have been developed – contrary to the examples used in the preceding two studies – independently from any actual technique. It is not clarified whether that constitutes a bias or not. However, while it was discussed in section 4.8.2 that no existing technique or tool for producing a DR-based documentation could meet all requisites to a satisfying degree, it is feasible to extend another technique (or tool) which produces documentation artifacts to contain such informally represented DR like the self-developed ones in those. Suggesting for that is, for example, the technique constituted by doc comments (see...
section 4.4). A textual DR could be represented by means of an appropriate doc tag. Further, the
decision and the rationale for it could each be implemented with a separate tag, even though
this has not been the case for the DR examples here, due to their representation in the form of
a continuous text. Interestingly, a documentation that contains DR obviously complies with the
personal expectations on documentation of a developer who was interviewed for obtaining gen-
eral information about HolidayCheck (see section 5.2.1). That participant – who did not take
part in the study series – casually uttered that in a documentation close to the code an explana-
tion why something has been implemented in a certain way was most important. This opinion
underpins the general helpfulness of a DR-based or, respectively, a DR-containing document-
tation – at least in the case of this specific software organization, and possibly also for other,
similar software organizations.

Like in the third study, the answers to the question about the general helpfulness of the consid-
ered documentation artifacts for a change impact analysis – here question #9.2 – are inverted
to those to analogous question #5.4 regarding the proportions of affirmative answers. Again,
this observation is not further clarified by the answers to the appendant free-text question #9.3.
But astonishingly, the answer to that by SD1 is completely identical with the exemplification
of DR in an agile context given in section 4.8.2. While this exemplification has not been sup-
posed to form a benefit in terms of the HIGHLEVELBENEFIT requisite there, the answer of
SD1 suggests that here. Indeed, it reveals a further direct benefit of the first type (see section
3.2): Obviously, an eliminated code refactoring – where it would be futile – means less effort
for the development or maintenance of a software product. It is essential for a direct benefit
that, in a concrete case, this effect is also discernible for project stakeholders by means of an
extracted documentation artifact. Certainly, the code refactoring is performed – or waived – by
a developer. However, it is easily conceivable that a project stakeholder could understand and
recognize the potential of a programming activity that would neither improve the code quality
nor save time in the development. This could be realized by explaining the project stakeholder
what code refactoring is and by using a marking like, for example, a keyword in a DR which
warns of doing code refactoring by justifying an unclean code part. Using such a keyword is
doubtlessly realizable, but also explaining the concept of a DR to a project stakeholder as code
refactoring was mentioned – and evidently understood by the participants – in the first study
while presenting aspects of agile software development that support program comprehension
(see section 5.5.1). Thus, a developer could recognize when the effort of maintaining or chang-
ing an appropriate “unclean” part of the code is reduced due to the futileness of continuous code
refactoring of that (see section 1).
6 Summary and discussion

Finally, the findings from both the theoretical part (see section 4) and the empirical part (see section 5) are summarized and discussed regarding the research questions (see section 1.4). This is split into two parts according to the research results – the developed research framework (section 6.1), and the obtained techniques and tools (section 6.2). The subsequent sections are devoted to discuss exclusively the scientific quality of both (section 6.3), the exceptional parts of the research process as well as the implications of those (section 6.4), and the value of the results for the scientific community as well as for practitioners (section 6.5). In the last section 6.6, motivations and possibilities for future research on the topic are elaborated.

6.1 Research framework

By deducing six requisites from mainly the research goals, the research framework has been developed. The intention to support program comprehension was taken into account by the LOWLEVELCONTEXT requisite and the PRIMARILYPROGRAMCOMPREHENSION requisite. The ARBITRARYAGILEMETHODOLOGY requisite and the ARBITRARYPROGRAMMINGLANGUAGE requisite were virtually an elaboration of the research goal that forms the essential conformity with agile software development regardless of any specific software organization. Also the HIGHLEVELBENEFIT requisite has already been largely embodied by the appropriate goal itself. However, like the NOSEPARATEARTIFACT requisite, its rationale reveals that two parties – developer stakeholders and project stakeholders – have to be addressed in the same way to combine both the documentation-based support of program comprehension and the conformity with agile software development. Innovativeness as a further research goal was not transformed into a requisite. It is neither required for the support of program comprehension nor for the conformity with agile software development. And it was also not required for limiting the scope of the thesis for any practical reasons. As mentioned, postulating it was simply a decision in favor of the novelty of this thesis. Because of that, it was not explicitly considered up to the empirical part of the thesis, where a selection of studied objects had to be done anyway due to limited resources.

The obtained techniques and tools that were treated as potential research results were discussed along the requisites. The HIGHLEVELBENEFIT requisite and the NOSEPARATEARTIFACT requisite proved to be generally the most problematic requisites in the case of these techniques and tools. The dichotomy between documentation and agile software development elucidated at the beginning of the thesis is substantiated by that finding.

Despite not constituting the main part in the survey studies, the research framework itself was also evaluated by the participants of the study series, which constituted the empirical part of the thesis. However, the framework could only be evaluated partly, as the ARBITRARYAGILEMETHODOLOGY requisite and the ARBITRARYPROGRAMMINGLANGUAGE requisite were seen as hardly relevant for a specific software organization. The results of this evaluation showed that all participants appraised the remaining four requisites as justified. Only the NOSEPARATEARTIFACT requisite could not convince all participants. In the appraisal of the

89
importance of each requisite regarding a participant’s involvement in agile software development, that requisite was, again, the least convincing requisite, followed by the PRIMARILYPROGRAMCOMPREHENSION requisite. Nonetheless, all requisites were largely appraised as important, either for the own involvement or for that of another participant. Not only those two less convincing requisites made clear that the participants had difficulties in comprehending the research framework – they also uttered that verbally. It remains an open question whether these difficulties occurred due to the scientific grounding of the framework or due to its form, which was at least regarded as irritating.

6.2 Techniques and tools

After an inspection of the found literature, aligned with the systematic narrowing down approach for searching for literature, six found techniques and tools were determined to be investigated further because they were considered as potential research results. For that, they were discussed along each requisite of the research framework. Apart from that, their documentation-based support of program comprehension and their conformity with agile software development was discussed. In this context, the decision to use scientifically less grounded literature only as additional references has not led to a deficient literature substantiation of the discussions. One technique – explaining the program logic through Literate Programming – was exclusively considered in the discussions for the sake of completeness and for providing a basis for later concepts of source code documentation. For the other five techniques and tools, it was deduced that all of them principally meet the requisites and thereby the research goals they base on. But evidently, none of them meet every requisite fully, as well as their support of program comprehension and their conformity with agile software development is not warranted in general. For most techniques and tools, constraints were specified to handle these deficiencies such that the research goals could still be regarded as achieved. However, also uncertainty proved to be present: For the HIGHLEVELBENEFIT requisite and the NOSeparateArtifact requisite, there were found two cases (see appendix C) where it was virtually unknown – solely through a literature review – whether they can be met at all. This uncertainty has been an additional motivation for doing consecutively empirical research.

All discussed tools violate the ARBITRARYPROGRAMMINGLANGUAGE requisite either by supporting only specific programming languages or – then in the broader sense – by implicating the use of certain software development tools. It is clear that it would be necessary to develop the tools further not only because of the prototype state of most of them but also to adapt them to the constraints in respect of meeting the ARBITRARYPROGRAMMINGLANGUAGE requisite. That would basically mean to extend the tools such that they support arbitrary OOP languages and that they do not prescribe a particular other tool. Thus, after the literature review it has already been evident that no immediately applicable tools are among the research results. Not revealed at that point in time was whether one of the techniques or tools is particularly promising in respect of the research goals.

In the study series, conducted after finishing the literature review, three of the techniques and tools were evaluated in practice with the purpose to verify those by the participants in respect of the HIGHLEVELBENEFIT requisite: generating scenario diagrams from unit tests, linking parts of the source code to user stories, and producing informally represented design rationales. Only
these three of five promising techniques and tools in total were considered because they were identified as the most innovative ones. Besides, such a limitation was also a necessity due to the limited resources for the studies. From the requisites, solely the **HIGHLEVELBENEFIT** requisite was of interest because it has already been a research question independently from the research framework itself what a directly induced benefit for project stakeholders could be. And because the **HIGHLEVELBENEFIT** requisite as the eventual embodiment of that goal proved to be – together with the **NOSEPARATEARTIFACT** requisite – more complex than the other requisites, focusing on it was even more suggesting. Consequently, the target population was adapted to that focus by an over-representation of project stakeholders considered in the sampling.

The presentation of the three chosen techniques and tools – in the case of the latter that of the fundamental technique – as part of three separate studies was understood by the participants to a large extent. The results obtained from the employed questionnaires proved that they all have grasped the concept behind each technique, what has been seen as a prerequisite for any interpretation of answers to other questions. One such question was whether the documentation artifacts resulting from the application of the techniques could be helpful for a change impact analysis conducted by project stakeholders. This question addressed the **HIGHLEVELBENEFIT** requisite – implicitly – as well as a concrete benefit which was supposed for all three techniques in the discussions of them in the literature review. Indeed, the documentation artifacts from all three techniques were appraised as helpful for a change impact analysis by many participants. However, it could not be clarified whether estimating the effort for an implementation of a change or whether estimating the risk potential involved in that – both essentially constitutes a change impact analysis – is facilitated.

The participants stated own ideas for further benefits. Most of those did not satisfy the notion of a benefit as specified by the **HIGHLEVELBENEFIT** requisite. Even so, there was one idea per technique that can feasibly result in such a benefit. The least vaguely stated idea was to use a design rationale exactly as exemplified in its introduction in the literature review: as a justification for a code part that is, as an exception, obfuscating – despite a regular compliance with code refactoring. Producing design rationales in natural language was also generally appraised by the participants as most helpful for a change impact analysis among all three techniques.

Furthermore, another employee of the organization uttered independently from the studies the idea to use a concept similar to a documentation based on design rationales for documenting source code. Therefore, it can be deduced that the documentation technique basing on informally represented design rationales is a particularly promising one.

### 6.3 Scientific quality

The virtually inevitable adherence to the perspective of a software developer during the research – anticipated in the introduction – was eventually confirmed through the statements by the participants of the survey studies. Consequently, it is a sign of scientific quality when the conducted research has adhered to the truth theory of software development as well. This criterion is evidently met as agile software development comes with a distinct empirical stance. And the conducted research was indeed empirical. Certainly, the literature review was counted as the theoretical part of the thesis, but the research works from which the techniques and tools were extracted have almost all comprised empirical research themselves. Thus, the thesis can
be regarded as considerably empirically grounded. For an appraisal of the particular quality of the own empirical research in the thesis, it is suggesting to use the criteria of Easterbrook et al. (2008) for empirical validity:

**Construct validity** (Are theoretical constructs interpreted and measured correctly?) The questionnaires in the studies comprised detailed specifications and delimitations in the formulation of each question. Furthermore, misinterpretations were actively prevented by comprehension questions and further verbal explanations.

**Internal validity** (Do the results really follow from the data?) Correlations in the data were investigated further to state a sound causation or at least a profound assumption. A statistical analysis of the data was only done to an extent that was seen as acceptable in respect of the sample size.

**External validity** (Are the claims for the generality of the results justified?) Choosing the research setting as well as the sampling strategy and methods was elaborated in great detail. In contrast to academic settings – prevailing in software engineering research – the eventual industrial setting is highly conform to the practice-oriented research purpose.

**Reliability** (Do the studies yield the same results if replicated?) Unfortunately, this criteria is definitely not fully satisfied as the eventual sample size was relatively small. Further, the participants had no extensive experience with agile software development. But at least the finding that the research framework is not optimally comprehensible for practitioners is not affected by that: It is a falsification, for which also a small sample size is sufficient, and particular experience with agile software development is obviously not required for evaluating comprehensibility in general.

A scientific grounding of the research framework and the techniques and tools as research results can be seen as fundamentally achieved because both clearly base on existing scientific knowledge. In this context, also theoretical additive has been contributed: The research framework was largely deduced from the agile values and principles, which are well-accepted in science as well as among software practitioners. But besides that, the requisites of the framework were substantiated with additional research works. All considered techniques and tools were referred to an unambiguous description of themselves, namely the research works from which they were extracted. The constraints in respect of meeting the requisites made in the discussions were partially accompanied by further research works. From a qualitative point of view, it can be concluded that the context of the existing techniques and tools was thus extended to the three research concepts of the thesis in combination. Hence, the research questions have largely been answered already in the theoretical part.

As a consequence of referring to previous research works to such an extent, the research framework and the techniques and tools become reproducible results – another criterion for scientific grounding. In the case of the techniques and tools, reproducibility is further facilitated by the fact that obtaining them has complied with known techniques by Oates (2006). Based on these, a systematic procedure for inspecting the found literature was developed and expounded. Likewise, the procedure for discussing a single technique or tool was presented. In the case of the research framework, it was not applied a specific procedure or method as this was not regarded as necessary – neither for the primary nor the secondary purpose of it. After all, the research framework has simply been a specification of the research goals from a qualitative point of
view. Nonetheless, it was expounded in great detail such that it is feasible to reproduce the development of it – for example, with modifications in the reasoning.

Worth mentioning in the context of scientific quality is also the consistency in the conduct of the research: Each technique or tool was treated in the same way – in the literature review as well as in the study series. Moreover, the abstraction of a tool by considering the fundamental technique implemented by it was applied consistently in both parts of the research. Finally regarding the quality of the eventual research results themselves, it can be stated what they are not: It is obvious that the obtained techniques and tools do not represent a comprehensive result as there can always be employed more research databases and keywords for a search. And certainly, there might be research works that are not published in a database at all. Although the research framework is clearly relevant in respect of the motivated research purpose, it does not lead to concrete technique or tools by itself. But it was also never supposed to do that.

6.4 Research process

Salient in the research process was the decision to implement the research goal about the innovativeness of a technique or tool not by transforming it into a requisite beforehand like the other research goals. Actually, this would have been feasible, for example, by deducing the justification for innovativeness from the ninth agile principle about technical excellence. Therefore this decision is indeed a rather arbitrary one without a proper rationale. Consequently, it can be seen as a flaw in the research design. However, it revealed as advantageous: To implement innovativeness as a research goal not earlier than in the survey studies has led to the consideration of more (non-innovative) techniques and tools – Literate Programming, doc comments, and unit tests. Even though those were eventually not considered as research results anymore, the findings from the discussions of them could be used in the discussions of the actually innovative techniques and tools for their further underpinning in respect of the research goals. This would not have been likewise suggesting if innovativeness had already been formulated as a requisite. Furthermore, the technique of doc comments provided a potential approach for an implementation of the concept of producing informally represented design rationales as a particularly promising documentation technique.

Not anticipated, the search for literature in the theoretical part casually brought several research works that could have been used for a further substantiation of the requisites. But such a use would have led to a tautology, because the requisites were intended as a framework for the techniques and tools proposed by exactly those research works and not vice versa. Certainly, if the framework had been intended to be used apart from any immediate internal purpose, for example, for a use in a later separate research work, then doing that would have been advisable. It can be assumed that new techniques and tools emerge continually – then not considered for the substantiation of the framework but for an evaluation by means of that. Contrary to that, rather anticipated was that there could occur problems in the design of the empirical part: The consecutive nature of it was accompanied with the uncertainty about the eventual object of research, which was supposed to be clarified not earlier than after the theoretical part. However, no discernible problems occurred. Choosing the HIGHLEVELBENEFIT requisite for further investigation and the sampling strategy as well as the used sampling methods was reasoned by pragmatic decisions – particularly regarding the available resources. Investigating those
three specific techniques was mainly predetermined by innovativeness as the still outstanding research goal. Hence, pragmatism as the philosophical stance underlying this thesis has clearly been implemented in the actual research.

The decision to exclude the ArbitraryAgileMethodology requisite and the ArbitraryProgrammingLanguage requisite in the study series – again a pragmatic one – is reasonable, because including them would have required a completely different sampling only for these two requisites. Otherwise, the occurrence of a sampling bias would have been to expect: Practitioners of a specific organization most probably do not appraise generality in respect of another organization of the same type as important. Anyway, it can be doubted whether it would not have been more advantageous to implement the appropriate research goal embodied by these two requisites likewise to the implementation of the innovativeness goal: without any transformation into one or several requisites and not earlier than in the empirical part. It is conceivable that, analogously to the consideration of non-innovative techniques and tools in the theoretical part, this could have led to further findings useful for the discussions of the actually considered techniques and tools. Nonetheless, it can be argued that an implementation in the form of requisites is still justified for the use of the framework in practice if it is not assumed that only practitioners from whom each belongs to a particular organization apply the framework. For example, an external consultant might regard the ArbitraryAgileMethodology requisite and the ArbitraryProgrammingLanguage requisite as crucial when consulting different organizations about choices for introducing documentation processes.

Definitely exceptional in the conducted research is the intentionally induced over-representation of project stakeholders through the sampling. In general, the target population included practitioners of different professions typically involved in the agile software development in a software organization – software developers, project managers, and product managers as representatives of the customer. This in consideration of collaboration, which is characteristic for an agile context. Hence, the relevance for practitioners of the findings from the studies has been strengthened. Further, choosing a particular software organization was also restricted beforehand to ensure a typical organization is chosen: It was determined that the organization where to conduct the studies had to be an SME company that develops web applications with an agile software development methodology. Finally, in the context of the research strategy for carrying out the studies there, it was additionally striven for conducting all survey studies in an agile manner analogously to the research setting. Certainly, the resulting sample and research setting might have been relatively unique but not exceptional in terms of abnormal: All these decisions were grounded sufficiently by appropriate theories. Only the over-representation of the project stakeholders was not done according to any well-proven scientific theory. Consequently, the validity of the findings is not warranted because of the realization of that decision, which is not scientifically grounded. Thus, the apparently fully satisfied validity of the survey studies – when considering the discussion of the validity criteria above – is certainly not optimally satisfied in this particular aspect.

6.5 Value for science and practice

Before this thesis, none of the chosen techniques have been empirically studied with such a restrictive and coherent set of research goals. Furthermore, most of the previous empirical studies
have not been done in an industrial research setting. Nonetheless, the particular novelty of this thesis is the unique topic of it. The presented research purpose and the research goals have concretized similar but vague approaches from previous research works. But it was also introduced a context – constituted by the combination of the three research concepts – that has not been considered up to now in the scientific community, at least not explicitly. For example, there might be research works that deal with documentation for the purpose of supporting program comprehension but that do not consider agile software development at all. Or there might be research works that deal with program comprehension in agile software development but that do not consider the documentation aspect. Therefore, this thesis provides a considerable contribution to the scientific community. The research framework was explicitly designed to be used for future research in that context. Not only to evaluate existing techniques or tools but possibly also to develop completely new ones. Besides techniques and tools for documenting for the purpose of supporting program comprehension in agile software development, also methods or processes for that purpose could eventually result from such research.

Certainly, the research framework represents also a valuable novelty in practice. But to warrant its applicability there, it obviously needs to be modified such that it is more comprehensible. Regardless whether the scientific grounding of it is causal of comprehension difficulties among practitioners or not, its form can definitely be improved by augmenting the structure of the requisites. Even though these were deduced from the research goals as obligatory specifications, considering them as proposals apart from their internal use in the thesis is feasible. And for that notion, there already exists a common form in the field of software development – a pattern – which is known for being constituted basically by a problem description and a solution for that problem (Gamma et al. 1995). Because a transformation of a requisite into such a structure seems to be possible and because it is more likely that a software practitioner comprehends the resulting pattern better than the requisite, a modification of the framework in favor of its applicability should be done by re-expressing it as a set of patterns. The work of Rüping (2003) demonstrated that patterns can be used in a context comprised by this thesis – documenting in agile software development.

Apparently, the value of the presented three techniques is primarily relevant for practitioners as they can build on those in their practical work. However, it is also relevant from a scientific point of view: First, the research purpose was evidently achieved by concretely demonstrating the possibility that agile values, principles, and practices are not fundamentally violated or impeded. Second, the techniques or, more general, the concepts of them are definitely worth investigating further in science as there are still unanswered questions. And it can taken as granted that these will not be sufficiently answered in practice as agile software development follows a strongly practice-oriented approach which implicates that such concepts are rather not put in a larger context than actually needed in a concrete case. As in science exactly that is done usually, further findings can potentially be gained by that, which might lead to inspirations for the enhancement of agile software development.

6.6 Future research

Some possibilities for conducting future research in the topic of the thesis have already been implicitly mentioned above. A possibility that should be mentioned explicitly by all means is
that of a more conclusive investigation of one of the three techniques: In the studies conducted for this thesis long-term implications of an actual application of a technique were not considered at all. It is worthwhile to do a case study in which a particular technique is productively employed to investigate exactly that. As producing design rationales in natural language revealed as the most promising technique, it should be focused on that. The suggestion to implement that technique by incorporating it into an existing tool raises the question whether and how this would influence the original support of program comprehension by that tool. This can also be a research question for future research.

An unanswered question in respect of the HIGHLEVELBENEFIT requisite, which evidently embodies a unique research goal, is likewise eligible to be studied in detail: Is particularly the estimation of the effort or particularly that of the risk potential facilitated? And of course, the direct benefits of the considered documentation artifacts as suggested by the participants of the studies can still be worked out more concretely. Thus, like the supposed facilitation of a change impact analysis, they can be verified with a survey study.

Eventually, repeating the study series conducted for this thesis in another software organization is also a recommendation for future research. This should be as well an SME company that develops web applications, but the sample size should be definitely larger in order to increase the reliability of the results.
References


URL: http://portal.acm.org/dl.cfm

URL: http://rmock.sourceforge.net/

URL: http://ieeexplore.ieee.org/servlet/opac?punumber=1000026


URL: http://agilemodeling.com/essays/agileDocumentation.htm

URL: http://agilemodeling.com/

URL: http://www.agilemodeling.com/artifacts/sequenceDiagram.htm


URL: http://www.atlassian.com/software/jira/

URL: http://mercurial.selenic.com/


**URL:** [http://agilemanifesto.org/](http://agilemanifesto.org/)

**URL:** [http://agilemanifesto.org/principles.html](http://agilemanifesto.org/principles.html)


**URL:** [http://www.mountaingoatsoftware.com/articles/35-advice-on-conducting-the-scrum-of-scrums-meeting](http://www.mountaingoatsoftware.com/articles/35-advice-on-conducting-the-scrum-of-scrums-meeting)


**URL:** [http://www.ctan.org/tex-archive/web/noweb/examples/](http://www.ctan.org/tex-archive/web/noweb/examples/)


URL: http://www.martinfowler.com/bliki/Xunit.html


URL: http://calla.ics.uci.edu:8080/Plone


URL: http://jpacman.sourceforge.net/


URL: http://books.google.com

URL: http://scholar.google.com


URL: http://www.holidaycheck.com/aboutus.php

URL: http://www.holidaycheck.com/


URL: http://www.program-comprehension.org

URL: http://conferences.computer.org/icsm/

URL: http://www.swebok.org

URL: http://ieeexplore.ieee.org


URL: http://download.java.net/jdk6/source/

URL: http://www.xprogramming.com/Practices/PracNotNeed.html


Kniberg, H. (2010), Kanban and Scrum - making the most of both, Lulu.com.


**URL:** [http://www_scrum.org/scrumguides/](http://www_scrum.org/scrumguides/)
Seacord, R. C., Plakosh, D. & Lewis, G. A. (2003), Modernizing Legacy Systems: Software Technologies, 
Engineering Process and Business Practices, Addison-Wesley Longman.


the Emergence of Design Rationale from Design Communication’, Artificial Intelligence for 


August 2010. 
URL: http://www.sigdoc.org/

Sim, S. E. (2010), ‘STICH – STory Integration and Tracking for Code comprHension’, Donald Bren 
School of Information and Computer Sciences, University of California, Irvine, CA, USA. Accessed 
23 August 2010. 
URL: http://calla.ics.uci.edu/stitch/stitch_main.html


Souza, V. E. S. & Falbo, R. (2005), An agile approach for web systems engineering, in ‘WebMedia 2005: 


URL: http://www.springerlink.com

Storey, M.-A. (2005), Theories, methods and tools in program comprehension: past, present and future, 

Cooperative Extension, University of Wisconsin-Extension, Madison, WI, USA.


Tilley, S. (2009), Documenting software systems with views VI: lessons learned from 15 years of re-
search & practice, in ‘SIGDOC ’09: Proceedings of the 27th ACM International Conference on Design 
of Communication’, ACM, pp. 239–244.

URL: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html


URL: http://www.infoq.com/articles/current-direction-of-agile


URL: http://www.stack.nl/dimitri/doxygen/


URL: http://ieeexplore.ieee.org/servlet/opac?punumber=1000635


URL: http://en.wikipedia.org/wiki/Aspect-oriented_programming


URL: http://en.wikipedia.org/wiki/Hello_world_program_examples
URL: http://en.wikipedia.org/wiki/Pac-Man


URL: http://en.wikipedia.org/wiki/Unique_visits

URL: http://onlinelibrary.wiley.com

Computer 36(6), 39–43.


Appendix A: Unrefactored unit test

Listing 5: The original – unrefactored – version of the unit test from listing 1

```java
public void testGetPageHierarchyAsXml() throws Exception {
    crawler.addPage(root, PathParser.parse("PageOne"));
crawler.addPage(root, PathParser.parse("PageOne.ChildOne"));
crawler.addPage(root, PathParser.parse("PageTwo"));

    request.setResource("root");
    request.addInput("type", "pages");
    Responder responder = new SerializedPageResponder();
    SimpleResponse response = (SimpleResponse) responder.makeResponse(
        new FitNesseContext(root), request);
    String xml = response.getContent();

    assertEquals("text/xml", response.getContentType());
    assertSubString("<name>PageOne</name>", xml);
    assertSubString("<name>PageTwo</name>", xml);
    assertSubString("<name>ChildOne</name>", xml);
}
```

```java
public void testGetPageHierarchyAsXmlDoesntContainSymbolicLinks() throws Exception {
    WikiPage pageOne = crawler.addPage(root, PathParser.parse("PageOne"));
crawler.addPage(root, PathParser.parse("PageOne.ChildOne"));
crawler.addPage(root, PathParser.parse("PageTwo"));

    PageData data = pageOne.getData();
    WikiPageProperties properties = data.getProperties();
    WikiPageProperty symLinks = properties.get(SymbolicPage.PROPERTY_NAME);
    symLinks.set("SymPage", "PageTwo");
    pageOne.commit(data);

    request.setResource("root");
    request.addInput("type", "pages");
    Responder responder = new SerializedPageResponder();
    SimpleResponse response = (SimpleResponse) responder.makeResponse(
        new FitNesseContext(root), request);
    String xml = response.getContent();

    assertEquals("text/xml", response.getContentType());
    assertSubString("<name>PageOne</name>", xml);
    assertSubString("<name>PageTwo</name>", xml);
    assertSubString("<name>ChildOne</name>", xml);
    assertNotSubString("SymPage", xml);
}
```

```java
public void testDataAsHtml() throws Exception {
    crawler.addPage(root, PathParser.parse("TestPageOne"), "test page");

    request.setResource("TestPageOne");
    request.addInput("type", "data");
    Responder responder = new SerializedPageResponder();
    SimpleResponse response = (SimpleResponse) responder.makeResponse(
        new FitNesseContext(root), request);
    String xml = response.getContent();

    assertEquals("text/html", response.getContentType());
    assertSubString("test page", xml);
    assertSubString("<Test", xml);
}
```

*taken from Martin (2009, pp. 125-126)*
Appendix B: Unabstracted scenario diagram

Figure 22: Unabstracted scenario diagram of a test case for *JPacman* opposed to the diagram in figure 5<sup>a</sup>

<sup>a</sup>taken from Cornelissen et al. (2007, sec. 5)
Appendix C: Meet/violate table technique or tool

The following table gives an overview over the meeting or violation of the requisites presented in section 3 for each of the techniques and tools summarized and discussed in section 4. Besides the requisites, the meeting or violation of agile software development in general is also listed, as in all the discussions appropriate requirements were directly deduced from that.

<table>
<thead>
<tr>
<th>Technique or tool</th>
<th>Requisite</th>
<th>LOW LEVEL</th>
<th>HIGH LEVEL</th>
<th>GREATLY</th>
<th>PROGRAM COMPREHENSION</th>
<th>ARBITRARY METHOD</th>
<th>ARBITRARY PROGRAMMING LANGUAGE</th>
<th>Agile software development in general</th>
</tr>
</thead>
<tbody>
<tr>
<td>Literate Programming</td>
<td>+</td>
<td>+_+</td>
<td>u</td>
<td>+</td>
<td>+_+</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Doc comments</td>
<td>+</td>
<td>+_+</td>
<td>+_+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>Unit tests</td>
<td>+</td>
<td>+_+</td>
<td>+_+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Scenario diagrams of unit tests</td>
<td>+_+</td>
<td>u</td>
<td>+_+</td>
<td>+</td>
<td>+</td>
<td>+_+</td>
<td>+_+</td>
<td>+</td>
</tr>
<tr>
<td>Links to user stories</td>
<td>+</td>
<td>+_+</td>
<td>+_+</td>
<td>+</td>
<td>+</td>
<td>+_+</td>
<td>+_+</td>
<td>+</td>
</tr>
<tr>
<td>Design rationales</td>
<td>+_+</td>
<td>+_+</td>
<td>+_+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
</tbody>
</table>

Table 2: Meet / violate table for each technique or tool

*a+: met by tool / technique
−: violated by tool / technique
+_+: met by tool / technique as constrained in the corresponding discussion
+_: violated by tool, but met by technique of that (only applicable in the case of a tool)
_u: virtually unknown whether met or violated by tool / technique

Appendix D: Patterns for Agile Documentation

The following table shows patterns for an Agile Documentation by Rüping (2003) in their unaltered Pattern Thumbnails form – a shortened version of them by the author himself, each consisting of a name, a description of a problem, and a possible solution.25

25Actually, this shortened version is an excerpt of the original version as the three covered components themselves are identical with their form in the original version. The latter is simply more detailed by comprising further components, e.g. forces associated with the problem.
<table>
<thead>
<tr>
<th>Name</th>
<th>Problem</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design Rationale</td>
<td>How can the team make sure that the foundations are laid for future design changes?</td>
<td>Design documents become a valuable source of information if they aren’t restricted to describing the actual design, but also focus on the rationale behind the design and explain why the particular design was chosen.</td>
</tr>
<tr>
<td>The Big Picture</td>
<td>How can people be introduced to a project without being confronted with a deluge of technical details?</td>
<td>A good feel for a project is best conveyed through a description of the ‘big picture’ of the architecture that underlies the system under construction.</td>
</tr>
<tr>
<td>Realistic Examples</td>
<td>How can abstract material be explained in a comprehensible way?</td>
<td>Project documents are much more convincing if they include realistic examples from the project’s context.</td>
</tr>
<tr>
<td>Document History</td>
<td>How can confusion be avoided between versions of a document?</td>
<td>A document history can explain the differences to previous versions of a document, and can relate the document to versions of the software it describes.</td>
</tr>
<tr>
<td>Document Landscape</td>
<td>How can team members get a good overview of what documentation exists in a project?</td>
<td>The project documentation can be represented as a kind of landscape that team members can use as a mental map when they retrieve or add information. A document landscape that roughly forms a tree suits human intuition best.</td>
</tr>
<tr>
<td>Code-Comment Proximity</td>
<td>What is an easy way to maintain documentation that refers to the actual code?</td>
<td>Documentation of the code, to the extent that a project team considers it necessary, is best done through source code comments. Separate documents should be reserved for higher-level issues such as overviews, requirements, design and architecture.</td>
</tr>
<tr>
<td>Single Source and Multiple Targets</td>
<td>How can multiple views of a document be created without doubling maintenance?</td>
<td>The documentation infrastructure can employ mechanisms that take source documents and automatically generate additional views. Such mechanisms avoid double maintenance and ensure consistency.</td>
</tr>
<tr>
<td>Notification upon Update</td>
<td>How can readers be prevented from using versions?</td>
<td>Whenever there is a significant change in a project document, all potential readers should be notified of the new version. The notification should roughly explain what has been changed, but should not include the updated material itself.</td>
</tr>
</tbody>
</table>

Table 3: *Pattern Thumbnails* for Agile Documentation

*a* taken in unchanged order from (Rüping 2003, pp. 197-204)
Appendix E: Questionnaires

QUESTIONNAIRE #1  Professional experience  Name:______________
STUDY 1

Please fill out this questionnaire, which comprises questions about your professional experience and education.

1. What is your position at HolidayCheck? Please state your job title at HolidayCheck for that. Prefixes like Senior or Junior as designations of your professional level are not relevant here.
   - Software Developer
   - Project Manager
   - Product Manager

2. In total, how long have you been working for HolidayCheck in the position as stated above or in another position that belongs to the same occupational field? For example, you might have done an internship in the software development department before being employed as software developer. In this case, the internship counts.
   - Less than 1 year
   - More than 1 year, less than 2 years
   - More than 2 years, less than 3 years
   - More than 3 years, less than 4 years
   - More than 4 years, less than 5 years
   - More than 5 years, less than 6 years
   - More than 6 years, less than 7 years
   - More than 7 years

3. In total, how long have you been working at all in the position as stated above or in another position that belongs to the same occupational field? Any professional experience (also internships) counts. It is not relevant here if a job title of a former occupation differs from the above-stated. For example, a previous occupation as a programmer counts, in the case you are a software developer now.
   - Less than 1 year
   - More than 1 year, less than 2 years
   - More than 2 years, less than 3 years
   - More than 3 years, less than 4 years
   - More than 4 years, less than 5 years
   - More than 5 years, less than 6 years
   - More than 6 years, less than 7 years
   - More than 7 years, less than 8 years
   - More than 8 years, less than 9 years
   - More than 9 years

4. Regarding your professional experience as stated above in question 3: In total, how long have you been actively involved as a stakeholder in agile software development? It is not relevant here which particular agile software development methodology, for example, XP or Scrum, has been applied.
   - Less than 1 year
   - More than 1 year, less than 2 years
   - More than 2 years, less than 3 years
   - More than 3 years, less than 4 years
   - More than 4 years, less than 5 years
   - More than 5 years

5. Do you have an education that is directly related to your position as stated above? For example, a degree in computer science can be considered as directly related to a position as software developer. It is not relevant here whether that education is a prerequisite for your position. Further, it is not relevant here whether you have any other education that is not directly related to your position.
   - Yes, at university/college level
   - Yes, not at university/college level
   - No
Please fill out this questionnaire, which comprises questions about the presented topic – documentation for the purpose of supporting program comprehension in agile software development. It has to be pointed out that these questions are of general nature. Thus, they are not particularly related to HolidayCheck or your occupation in this software organization.

1. Do you think that agile software development generally comes with an increased need for supporting program comprehension, compared to "non-agile" software development? An example for the latter is the waterfall model as a carried out software development process.
   - Yes
   - Rather yes
   - Rather no
   - No
   - I am not sure

2. If you want to comment your answer to question 1 or any difficulties in understanding it, then feel free to do that here:

3. Do you think that program comprehension can be supported with documentation such that it does not contradict agile values and principles? An example for an agile value is "Working software over comprehensive documentation". An example for an agile principle is "Simplicity – the art of maximizing the amount of work not done – is essential". Documentation here is any textual or graphical artifact that you consider as potentially producible by software developers.
   - Yes
   - Rather yes
   - Rather no
   - No
   - I am not sure

4. If you want to comment your answer to question 3 or any difficulties in understanding it, then feel free to do that here:
Please fill out this questionnaire, which comprises questions about the presented requisites. These requisites have to be met by any documentation technique or tool potentially leading to a support of program comprehension in agile software development. It has to be pointed out that these questions are particularly related to HolidayCheck and your occupation in this software organization.

1. Do you think that the following requisite is justified in respect of a conceivable application of it in the agile software development at HolidayCheck?
   (a) **LowLevelContext** requisite
   ☐ Yes ☐ Rather yes ☐ Rather no ☐ No ☐ I am not sure
   (b) **HighLevelBenefit** requisite
   ☐ Yes ☐ Rather yes ☐ Rather no ☐ No ☐ I am not sure
   (c) **NoSeparateArtifact** requisite
   ☐ Yes ☐ Rather yes ☐ Rather no ☐ No ☐ I am not sure
   (d) **PrimarilyProgramComprehension** requisite
   ☐ Yes ☐ Rather yes ☐ Rather no ☐ No ☐ I am not sure

2. If you want to comment any of your answers to questions 1a-1d or any difficulties in understanding them, then feel free to do that here:

3. Do you think that the following requisite is of importance for your involvement in the agile software development at HolidayCheck? Consider only your personal involvement, not that of any other stakeholder.
   (a) **LowLevelContext** requisite
   ☐ Yes ☐ Rather yes ☐ Rather no ☐ No ☐ I am not sure
   (b) **HighLevelBenefit** requisite
   ☐ Yes ☐ Rather yes ☐ Rather no ☐ No ☐ I am not sure
   (c) **NoSeparateArtifact** requisite
   ☐ Yes ☐ Rather yes ☐ Rather no ☐ No ☐ I am not sure
   (d) **PrimarilyProgramComprehension** requisite
   ☐ Yes ☐ Rather yes ☐ Rather no ☐ No ☐ I am not sure

4. If you want to comment any of your answers to questions 3a-3d or any difficulties in understanding
5. Do you think that the following requisite is of importance for the involvement of other stakeholders in the agile software development at HolidayCheck? Consider all stakeholders that could be involved.

(a) **LOWLEVELCONTEXT** requisite

- [ ] Yes
- [ ] Rather yes
- [ ] Rather no
- [ ] No
- [ ] I am not sure

(b) **HIGHLEVELBENEFIT** requisite

- [ ] Yes
- [ ] Rather yes
- [ ] Rather no
- [ ] No
- [ ] I am not sure

(c) **NOSEPARATEARTIFACT** requisite

- [ ] Yes
- [ ] Rather yes
- [ ] Rather no
- [ ] No
- [ ] I am not sure

(d) **PRIMARILYPROGRAMCOMPREHENSION** requisite

- [ ] Yes
- [ ] Rather yes
- [ ] Rather no
- [ ] No
- [ ] I am not sure

6. If you want to comment any of your answers to questions 5a-5d or any difficulties in understanding them, then feel free to do that here:
Please fill out this questionnaire, which comprises questions about sequence diagrams.

1. Have you grasped the concept of sequence diagrams in principle? It is not important that you have understood each detail.
   - Yes
   - Rather yes
   - Rather no
   - No
   - I am not sure

2. Do you basically understand this sequence diagram, which was shown in the presentation? I.e. can you interpret the objects and the messages between these for the most part?
   - Yes
   - Rather yes
   - Rather no
   - No
   - I am not sure

3. Do you understand any essential part of this sequence diagram, which depicts a certain run of an implementation of the video game Pac-Man? I.e. can you recognize any player interaction that you know from video games?
   - Yes
   - Rather yes
   - Rather no
   - No
   - I am not sure
4. Compared with the sequence diagram from the previous question, do you consider this sequence diagram—likewise depicting a certain run of the same implementation of the game—as less understandable? i.e. is it harder for you to recognize again the interaction that you might have recognized before?
1. Have you grasped the concept of a change impact analysis in principle? It is not important that you have understood each detail.

☐ Yes   ☐ Rather yes   ☐ Rather no   ☐ No   ☐ I am not sure

2. Do you consider the sequence diagram from question 2 of the previous questionnaire (#4) as helpful for a change impact analysis conducted by you? For that, imagine to conduct the analysis for a change request like, for example, adding a simple “change password” function.

☐ Yes   ☐ Rather yes   ☐ Rather no   ☐ No   ☐ I am not sure

3. Compared with the previous question, do you consider the sequence diagram from question 3 of the previous questionnaire (#4) as likewise helpful for a change impact analysis conducted by you? For that, imagine to conduct the analysis for a comparable change request.

☐ Yes   ☐ Rather yes   ☐ Rather no   ☐ No   ☐ I am not sure

4. Finally, do you consider sequence diagrams like the ones from the previous questionnaire (#4) as generally helpful for conducting a change impact analysis?

   (a) For estimating the effort

☐ Yes   ☐ Rather yes   ☐ Rather no   ☐ No   ☐ I am not sure

(b) For estimating the risk potential

☐ Yes   ☐ Rather yes   ☐ Rather no   ☐ No   ☐ I am not sure

5. Regarding the previous question, can you imagine any further direct benefits of sequence diagrams for you apart from a helpfulness for a change impact analysis? If yes, please describe that here:
Please fill out this questionnaire, which comprises questions about links from source code to user stories.

1. Have you grasped the concept of links from source code to user stories in principle? It is not important that you have understood each detail.
   - [ ] Yes
   - [ ] Rather yes
   - [ ] Rather no
   - [ ] No
   - [ ] I am not sure

2. Do you basically understand these visualizations of the links, which were shown in the presentation? I.e. can you recognize the links and can you identify the abstraction level of their visualization? An example for the abstraction level is the code line level.
   - (a) [ ] Yes
   - [ ] Rather yes
   - [ ] Rather no
   - [ ] No
   - [ ] I am not sure
   - (b) [ ] Yes
   - [ ] Rather yes
   - [ ] Rather no
   - [ ] No
   - [ ] I am not sure
   - (c) [ ] Yes
   - [ ] Rather yes
   - [ ] Rather no
   - [ ] No
   - [ ] I am not sure
Please fill out this questionnaire, which comprises questions about a change impact analysis based on links from source code to user stories.

1. Do you consider the visualizations from question 2 of the previous questionnaire (#6) as helpful for a change impact analysis conducted by you? For that, imagine to conduct the analysis for a change request like, for example, considering Blu-ray Discs as data storage media in addition to DVDs.
   - Yes
   - Rather yes
   - Rather no
   - No
   - I am not sure

2. Do you consider visualizations of links from source code to user stories similar to those from the previous questionnaire (#6) as generally helpful for conducting a change impact analysis?
   (a) For estimating the effort
   - Yes
   - Rather yes
   - Rather no
   - No
   - I am not sure
   (b) For estimating the risk potential
   - Yes
   - Rather yes
   - Rather no
   - No
   - I am not sure

3. Regarding the previous question, can you imagine any further direct benefits of such visualizations for you apart from a helpfulness for a change impact analysis? If yes, please describe that here:
Please fill out this questionnaire, which comprises questions about design rationales.

1. Have you grasped the concept of design rationales in principle? It is not important that you have understood each detail.
   - [ ] Yes
   - [ ] Rather yes
   - [ ] Rather no
   - [ ] No
   - [ ] I am not sure

2. Do you basically understand these design rationales, which were shown in the presentation? I.e. can you recognize the decision as well as its rationale? It is not important that you completely understand the decision itself.
   (a) The user name is not case-sensitive. However, in the authentication process case sensitivity is considered for the error message in the case of a failed login due to an incorrect user name. In this message the incorrect user name is re-stated. If that were lowercased by the system in order to process it further, the user could be irritated because the user name might have been entered case-sensitively. Therefore the user name is represented here by two variables. One holds the user name as originally entered, the other one as used internally.
      - [ ] Yes
      - [ ] Rather yes
      - [ ] Rather no
      - [ ] No
      - [ ] I am not sure

   (b) In the authentication process the password is passed together with the date when it has been set such that it can be determined to encrypt it either with MD5 or with SHA-2. This is necessary for enabling the matching with the hash stored in the database. Passwords that have been set after 2008-10-01 were stored in the database encrypted with SHA-2 instead of with MD5, because MD5 is not considered secure anymore. As an alternative to that the obsolete MD5 hashes could have been replaced by SHA-2 hashes. But doing so would have required to re-set the affected passwords because they cannot be retrieved from their hashes.
      - [ ] Yes
      - [ ] Rather yes
      - [ ] Rather no
      - [ ] No
      - [ ] I am not sure
1. Do you consider the two design rationales from question 2 of the previous questionnaire (#8) as helpful for a change impact analysis conducted by you? For that, imagine to conduct the analysis for a change request like, for example, adding a simple "change password" function.

☐ Yes ☐ Rather yes ☐ Rather no ☐ No ☐ I am not sure

2. Do you consider design rationales similar to those from the previous questionnaire (#8) as generally helpful for conducting a change impact analysis?
   (a) For estimating the effort
   ☐ Yes ☐ Rather yes ☐ Rather no ☐ No ☐ I am not sure
   (b) For estimating the risk potential
   ☐ Yes ☐ Rather yes ☐ Rather no ☐ No ☐ I am not sure

3. Regarding the previous question, can you imagine any further direct benefits of such design rationales for you apart from a helpfulness for a change impact analysis? If yes, please describe that here:
Appendix F: Likert-scale questions results

<table>
<thead>
<tr>
<th>Question</th>
<th>SD1</th>
<th>SD2</th>
<th>SD3</th>
<th>PJM1</th>
<th>PJM2</th>
<th>PDM1</th>
<th>PDM2</th>
<th>PDM3</th>
</tr>
</thead>
<tbody>
<tr>
<td>#2.1</td>
<td>y</td>
<td>(y)</td>
<td>(y)</td>
<td>n</td>
<td>y</td>
<td>n</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>#2.3</td>
<td>(y)</td>
<td>(y)</td>
<td>(y)</td>
<td>(y)</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>(y)</td>
</tr>
<tr>
<td>#3.1a</td>
<td>y</td>
<td>(y)</td>
<td>-</td>
<td>y</td>
<td>-</td>
<td>y</td>
<td>y</td>
<td>(y)</td>
</tr>
<tr>
<td>#3.1b</td>
<td>(y)</td>
<td>(y)</td>
<td>-</td>
<td>(y)</td>
<td>y</td>
<td>(y)</td>
<td>y</td>
<td>(y)</td>
</tr>
<tr>
<td>#3.1c</td>
<td>(y)</td>
<td>y</td>
<td>-</td>
<td>(y)</td>
<td>(n)</td>
<td>y</td>
<td>(n)</td>
<td>(n)</td>
</tr>
<tr>
<td>#3.1d</td>
<td>y</td>
<td>(y)</td>
<td>-</td>
<td>(y)</td>
<td>y</td>
<td>(y)</td>
<td>y</td>
<td>(y)</td>
</tr>
<tr>
<td>#3.1e</td>
<td>(n)</td>
<td>(y)</td>
<td>0</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>#3.3c</td>
<td>(n)</td>
<td>y</td>
<td>-</td>
<td>-</td>
<td>(y)</td>
<td>n</td>
<td>(y)</td>
<td>(n)</td>
</tr>
<tr>
<td>#3.3d</td>
<td>y</td>
<td>(y)</td>
<td>-</td>
<td>(y)</td>
<td>(n)</td>
<td>-</td>
<td>(y)</td>
<td>-</td>
</tr>
<tr>
<td>#3.4a</td>
<td>(n)</td>
<td>y</td>
<td>-</td>
<td>y</td>
<td>(n)</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>#3.4b</td>
<td>(n)</td>
<td>y</td>
<td>-</td>
<td>y</td>
<td>(y)</td>
<td>y</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>#3.5a</td>
<td>-</td>
<td>(y)</td>
<td>-</td>
<td>y</td>
<td>(y)</td>
<td>(n)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>#3.5b</td>
<td>-</td>
<td>(y)</td>
<td>-</td>
<td>y</td>
<td>(y)</td>
<td>(n)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>#3.5c</td>
<td>-</td>
<td>(y)</td>
<td>-</td>
<td>y</td>
<td>(y)</td>
<td>(n)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>#3.5d</td>
<td>-</td>
<td>(y)</td>
<td>-</td>
<td>y</td>
<td>(y)</td>
<td>(n)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>#3.6a</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>#3.6b</td>
<td>(y)</td>
<td>y</td>
<td>y</td>
<td>(y)</td>
<td>y</td>
<td>(y)</td>
<td>y</td>
<td>(y)</td>
</tr>
<tr>
<td>#3.7a</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>#3.7b</td>
<td>(y)</td>
<td>(y)</td>
<td>(y)</td>
<td>(y)</td>
<td>(y)</td>
<td>(n)</td>
<td>(n)</td>
<td>(y)</td>
</tr>
<tr>
<td>#3.8a</td>
<td>(n)</td>
<td>(y)</td>
<td>(y)</td>
<td>(n)</td>
<td>-</td>
<td>y</td>
<td>(n)</td>
<td>(n)</td>
</tr>
<tr>
<td>#3.8b</td>
<td>(n)</td>
<td>(y)</td>
<td>(y)</td>
<td>(y)</td>
<td>(n)</td>
<td>n</td>
<td>(y)</td>
<td>(n)</td>
</tr>
<tr>
<td>#3.9a</td>
<td>(n)</td>
<td>(y)</td>
<td>(y)</td>
<td>(n)</td>
<td>(y)</td>
<td>(n)</td>
<td>(y)</td>
<td>(y)</td>
</tr>
<tr>
<td>#3.9b</td>
<td>(n)</td>
<td>(n)</td>
<td>(n)</td>
<td>(y)</td>
<td>(n)</td>
<td>(y)</td>
<td>(n)</td>
<td>(y)</td>
</tr>
</tbody>
</table>

Table 4: Results of the Likert-scale questions from questionnaires #2 to #9

*"y" : "yes"   (y) : "rather yes"   (n) : "rather no"   n : "no"   - : "I am not sure"*
University of Borås is a modern university in the city center. We give courses in business administration and informatics, library and information science, fashion and textiles, behavioral sciences and teacher education, engineering and health sciences.

In the School of Business and Informatics (IDA), we have focused on the students' future needs. Therefore we have created programs in which employability is a key word. Subject integration and contextualization are other important concepts. The department has a closeness, both between students and teachers as well as between industry and education.

Our courses in business administration give students the opportunity to learn more about different businesses and governments and how governance and organization of these activities take place. They may also learn about society development and organizations' adaptation to the outside world. They have the opportunity to improve their ability to analyze, develop and control activities, whether they want to engage in auditing, management or marketing.

Among our IT courses, there's always something for those who want to design the future of IT-based communications, analyze the needs and demands on organizations' information to design their content structures, integrating IT and business development, developing their ability to analyze and design business processes or focus on programming and development of good use of IT in enterprises and organizations.

The research in the school is well recognized and oriented towards professionalism as well as design and development. The overall research profile is Business-IT-Services which combine knowledge and skills in informatics as well as in business administration. The research is profession-oriented, which is reflected in the research, in many cases conducted on action research-based grounds, with businesses and government organizations at local, national and international arenas. The research design and professional orientation is manifested also in InnovationLab, which is the department's and university's unit for research-supporting system development.