Supporting Product Development with Software from the Bazaar

Klaas-Jan Stol

UNDER JOINT SUPERVISION OF:
Dr Muhammad Ali Babar
Professor Paris Avgeriou
Professor Brian Fitzgerald

A DISSERTATION SUBMITTED IN FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

SUBMITTED TO THE UNIVERSITY OF LIMERICK
NOVEMBER 2011
Supervisors:  Dr Muhammad Ali Babar
Professor Paris Avgeriou
Professor Brian Fitzgerald

Examiners:  Dr Gabriela Avram (chair)
Dr Brian Lings (external)
Dr John Noll (internal)
The research presented in this doctoral dissertation is partially funded by the Irish Research Council for Science, Engineering and Technology (IRCSET) under grant number RS/2008/134, and by Science Foundation Ireland (SFI) grant number 03/CE2/I202_1 to Lero—The Irish Software Engineering Research Centre (www.lero.ie).
To my parents,

Herman & Ali,

for all their support.
Contact Information

Readers of this dissertation are welcome to contact the author at the following email address if they request any further information or clarification:

Klaas-Jan Stol
kjstol@gmail.com
# Table of Contents

- **List of Figures** ................................................................. xiii
- **List of Tables** ................................................................. xv
- **List of Abbreviations** ..................................................... xviii
- **Acknowledgements** ....................................................... xix
- **Summary** ........................................................................... xxi
- **Note** .................................................................................. xxii

### Chapter 1 Introduction .......................................................... 1
  1.1 **INTRODUCTION** .............................................................. 1
  1.2 **RESEARCH OBJECTIVE** .................................................... 3
  1.3 **RESEARCH OVERVIEW AND CONTRIBUTIONS** .................. 4
  1.4 **STRUCTURE OF THE THESIS** .......................................... 9

### Chapter 2 The Emergence of Open Source Software ............... 15
  2.1 **INTRODUCTION AND CHAPTER LAYOUT** ....................... 15
  2.2 **ORGANISATIONAL ADOPTION OF OPEN SOURCE SOFTWARE** 15
      2.2.1 Adoption of OSS development tools .................................. 17
      2.2.2 Integrating OSS components in product development ............ 18
      2.2.3 Using OSS development practices ...................................... 18
      2.2.4 Opensourcing a product ..................................................... 18
      2.2.5 Participating in OSS development ...................................... 19
  2.3 **COMPARISON OF OPEN SOURCE SOFTWARE AND INNER SOURCE SOFTWARE INTEGRATION** .................................................. 19
  2.4 **OPEN SOURCE DEVELOPMENT PRACTICES** .......................... 21
      2.4.1 The Open Source Software development community ............ 22
      2.4.2 Open Source Software development practices ....................... 25
      2.4.3 The Open Source paradigm as an agile method ..................... 30
  2.5 **CHAPTER SUMMARY** ..................................................... 31

### Chapter 3 The Emergence of Bazaar-Style Software Development in Cathedrals ......................................................... 33
  3.1 **INTRODUCTION AND CHAPTER LAYOUT** .......................... 33
  3.2 **MOTIVATION TO ADOPT INNER SOURCE** ........................... 33
  3.3 **INNER SOURCE ADOPTION MODELS** ............................... 35
      3.3.1 Infrastructure-based Inner Source model ............................ 35
      3.3.2 Project-based Inner Source model ...................................... 36
      3.3.3 Comparing infrastructure- and project-based Inner Source .... 37
  3.4 **THE ADOPTION OF OSSD PRACTICES IN INDUSTRY** ............. 40
  3.5 **NORMATIVE GUIDELINES TO ADOPT INNER SOURCE** ............ 43
      3.5.1 Raymond’s preconditions for bazaar style development ......... 43
      3.5.2 Lessons learnt by Neus and Scherf ................................. 43
      3.5.3 Sharma et al.’s framework .................................................. 44
      3.5.4 Adopting OSSD practices through adopting OSSD tools ........ 47
      3.5.5 Torkar et al.’s opportunities to adopt OSSD practices .......... 49
      3.5.6 Gurbani et al.’s suggested software characteristics ................ 52
  3.6 **OVERVIEW OF GUIDELINES** ............................................ 52
Chapter 12 Summary and Future Work

12.1 INTRODUCTION AND CHAPTER LAYOUT ................................................................. 299
12.2 REVIEW OF THE RESEARCH OBJECTIVES ................................................................ 299

12.2.1 Research questions for objective 1: Open Source ............................................ 300
12.2.2 Research questions for objective 2: Inner Source ............................................ 301

12.3 CONTRIBUTIONS AND IMPLICATIONS FOR RESEARCH AND PRACTICE ...... 302

12.3.1 RQ1: Challenges in OSS ............................................................................... 302
12.3.2 RQ2: OSS Evaluation methods ...................................................................... 303
12.3.3 RQ3: Architectural Knowledge needs of OSS integrators ............................... 304
12.3.4 RQ4: Identifying Architectural Patterns in OSS ............................................ 304
12.3.5 RQ5: Challenges in Inner Source .................................................................. 305
12.3.6 RQ6: Key factors in adopting Inner Source ................................................... 305

12.4 FUTURE WORK ....................................................................................................... 306

12.4.1 Future Work in Research on Using OSS products ......................................... 306
12.4.2 Future Work in Research on Inner Source .................................................... 308

Bibliography .................................................................................................................. 311

Appendix A Interview Guide used in Survey (Chapter 8) ......................................... 329

Appendix B Interview Guide used in Improvement Paradigm (Chapter 9) ................. 331

Appendix C Interview Guide used in Case Study SoftCom (Chapter 10) ................. 333

Appendix D Interview Guide used in Case Study NewCorp (Chapter 11) .................. 335
List of Figures

Figure 1-1. Overview of studies and publications. ........................................ 6

Figure 1-2. Contributions of this thesis, categorised along dimensions of bazaar type and contribution type. ...................................................... 8

Figure 1-3. Structure of this thesis. .......................................................... 10

Figure 2-1. Different ways for organisations to benefit or adopt from OSS. Arrows indicate equivalence or similarity. Grey boxes group equivalent options. ........................................................................... 17

Figure 2-2. Comparison of integration scenarios: Left: Integration of OSS components. Right: Integration of Inner Source Software components. .......................................................................................... 20

Figure 2-3. The onion model representing the community structure of a typical OSS project (taken from Crowston et al. 2004). .................. 23

Figure 3-1. The 3 tiers of Progressive Open Source (taken from (Dinkelacker et al., 2002)). .............................................................. 35

Figure 4-1. Classification of components based on where and how they are developed (in UML 2 notation).................................................. 59

Figure 4-2. Extended architecture life cycle, adapted from Tang et al. (2010). .............................................................................................. 62

Figure 5-1. Punch’s inductive-deductive hierarchy (based on a visualisation by (Leshem and Trafford, 2007). .................. 71

Figure 5-2. Prestructured versus unfolding research (from (Punch, 2006)). .................................................................................................. 73

Figure 5-3. Overview of the research design. .............................................. 76

Figure 5-4. The choice of research method is affected by several factors, as argued by different authors. ......................................................... 77

Figure 5-5. Improvement Paradigm (adapted from (Parra et al., 1997)). 92

Figure 5-6. Overview of the research questions, research methods and data collection methods used in this thesis. ................................. 97

Figure 6-1. Recall and precision depend on the outcome (oval) of a query and its relation to all relevant results (left-hand side of the oval) and
the non-relevant results (right-hand side of the oval). (Figure adapted from: Wikipedia). ................................................................. 103

Figure 6-2. Search approach. The review study was designed to identify all papers that studied OSS (phase 1). From this repository, sub-selections were made on specific topics (phase 2)........................... 105

Figure 6-3. Overview of the study selection procedure. ...................... 108

Figure 9-1. Overview of the three iterations of the Improvement Paradigm. .......................................................................................... 169

Figure 9-2. Timeline of studies conducted for this research ................. 170

Figure 9-3. Initial design of IDAPO. ..................................................... 186

Figure 9-4. Enhanced version of IDAPO (v2) in Business Process Modelling Language (BPMN) notation. ................................. 193

Figure 9-5. Distribution of numbers of identified patterns by the control group, treatment group (candidate and final). Boxplots 1-3: counting all patterns, corresponding to columns 1-3 in Table 9-8. Boxplots 4-6: trusted patterns only, corresponding to columns 4-6 in Table 9-8. ................................................................. 201

Figure 10-1. Example of “box-and-line” visualisation. ......................... 220

Figure 10-2. Conceptual model of Inner Source in the SoftCom organisation. Arrows between actors, products or processes indicate the order of reading, e.g., a Business division Participates in Collaborative development. ............................................................... 222

Figure 10-3. Mapping of the challenges identified in the literature, the case study, and SoftCom’s approaches to address them. Challenges coloured dark grey do not have associated approaches to address them. Challenges coloured light grey were identified in the case study but were not previously identified in the literature as a challenge with using OSS. ................................................................. 238

Figure 11-1. Relationships (numbered a to i) among the 4 categories of the framework for analysing organisations’ suitability for adopting Inner Source. ............................................................................................... 261

Figure 11-2. Overview of how machines of product lines A and B relate. The visited location develops and manufactures machines of product line A................................................................. 278
List of Tables

Table 3-1. Roles in the core team, as identified by Gurbani et al. (2010). ................................................................. 37

Table 3-2. Comparison of Infrastructure-based and project-based Inner Source models. ............................................. 38

Table 3-3. Overview of reported cases of Inner Source adoption............ 41

Table 3-4. Raymond’s preconditions for bazaar-style development........ 43

Table 3-5. Lessons learnt to introduce open source collaboration methods, according to Neus and Scherf (2005). ............ 44

Table 3-6. Transition steps to hybrid-OSS according to Sharma et al. (2002) ................................................................. 45

Table 3-7. Framework for creating a hybrid-OSS community, according to Sharma et al. (2002). ........................................ 46

Table 3-8. OSS development practices and their tensions with traditional approaches, according to Robbins (2005) ............ 48

Table 3-9. Impact of adopting OSS development tools, according to Robbins (2005) .......................................................... 49

Table 3-10. Guidelines by Torkar et al. (2011). ........................................ 50

Table 3-11. Features of suitable projects for an open source development style, according to Gurbani et al. (2006). ............... 52

Table 3-12. Overview of categories of guidelines proposed by the various authors. .............................................................. 53

Table 4-1. Mapping of phases of the architecture life cycle. ............... 65

Table 5-1. Research approaches and their usage (from Glass et al., 2002). .................................................................... 78

Table 5-2. Research methods and their usage (from Glass et al., 2002). .................................................................... 80

Table 5-3. Classification of study purposes (according to Marshall and Rossman) ............................................................ 82

Table 5-4. Levels of maturity and types of research questions (according to Edmondson and McManus). ................................. 82
Table 9-3. Challenges encountered by participants during the identification of architectural patterns in OSS products. 

Table 9-4. Suggestions of participants for improving the pattern identification process.

Table 9-5. Participants of the experiment.

Table 9-6. Participants’ self-assessed levels of expertise.

Table 9-7. Derivation of a trusted subset of patterns.

Table 9-8. Number of patterns per group, mean and standard deviation. Columns 1-3 consider all patterns identified; columns 4-6 only consider the trusted patterns.

Table 9-9. Precision and recall for control, treatment candidate and treatment final results.

Table 9-10. Hypotheses and resulting p-values of the Mann-Whitney U test.

Table 9-11. Post-study questionnaire results for the treatment group. High scores are highlighted.

Table 9-12. Post-study questionnaire results for the control group.

Table 10-1. Participants in the case study at SoftCom.

Table 10-2. Challenges identified in the case study and references to challenges related to the use of OSS products as identified in the literature.

Table 10-3. Overview of relevance of challenges to OSS, infrastructure-based Inner Source and project-based Inner Source.

Table 11-1. Details of the interviewed members of staff at NewCorp.

Table 11-2. Framework for assessing compatibility of Inner Source at organisations.

Table 11-3. Potentially incompatible practices identified by Gurbani et al. and how they map to the categories identified by the researcher.

Table 11-4. Components developed at NewCorp that may be candidates for Inner Source projects.

Table 11-5. Analysis of candidate components at NewCorp along the elements of the software product category of the analysis framework.

Table 11-6. Findings from case study and tension points that may arise in adopting Inner Source.
# List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>AK</td>
<td>Architectural Knowledge</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>BPMN</td>
<td>Business Process Modelling Notation</td>
</tr>
<tr>
<td>COS</td>
<td>Corporate Open Source</td>
</tr>
<tr>
<td>COTS</td>
<td>Commercial Off-The-Shelf</td>
</tr>
<tr>
<td>FOCOSEM</td>
<td>Framework for Comparing Open Source Evaluation Methods</td>
</tr>
<tr>
<td>GPL</td>
<td>General Public License</td>
</tr>
<tr>
<td>IDAPO</td>
<td>process for Identifying Architectural Patterns in OSS</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>IPR</td>
<td>Intellectual Property Rights</td>
</tr>
<tr>
<td>ISS</td>
<td>Inner Source Software</td>
</tr>
<tr>
<td>ISSD</td>
<td>Inner Source Software Development</td>
</tr>
<tr>
<td>OSS</td>
<td>Open Source Software</td>
</tr>
<tr>
<td>OSSD</td>
<td>Open Source Software Development</td>
</tr>
<tr>
<td>POS</td>
<td>Progressive Open Source</td>
</tr>
<tr>
<td>POSA</td>
<td>Pattern-Oriented Software Architecture</td>
</tr>
<tr>
<td>SA</td>
<td>Software Architecture</td>
</tr>
<tr>
<td>SLR</td>
<td>Systematic Literature Review</td>
</tr>
<tr>
<td>SMS</td>
<td>Systematic Mapping Study</td>
</tr>
<tr>
<td>SPL</td>
<td>Software Product Line</td>
</tr>
<tr>
<td>SR</td>
<td>Systematic Review</td>
</tr>
<tr>
<td>UML</td>
<td>Universal Modelling Language</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
</tbody>
</table>
Acknowledgements

There are many people that I would like to thank for their support during this research. First and foremost, I wish to thank my three supervisors, all of which have been extensively involved at different points of this research. Firstly, Professor Paris Avgeriou, who got me into doing this PhD in the first place. Thanks for your continuous support and encouragement. Paris also facilitated much of the research I’ve done in the Netherlands, and accommodated me with office space during my stays in Groningen. Secondly, I thank Dr Muhammad Ali Babar, who offered me the position in Ireland and guided me extensively during this research, in particular during the first 18 months or so. Ali also worked with me on many a weekend finishing papers. Thirdly, I wish to thank Professor Brian Fitzgerald, who was often available for insightful advice and guidance related to the research as well as enjoyable stories, anecdotes and tricks of the trade. I especially appreciated and enjoyed the “30 minute meetings” that became three-hour sessions. Furthermore, I am also grateful to the members of my defence committee, and in particular Dr Brian Lings and Dr John Noll, whose comments have improved the quality of this thesis.

I’m grateful to the Irish Research Council for Science, Engineering and Technology (IRCSET) who provided me with a research scholarship, and to Lero—the Irish Software Engineering Research Centre, who have also contributed financial support that allowed me to conduct this research and travel to various conferences over the world.

I received much support and help from my colleagues, for which I’m very grateful. Thanks to everyone in Lero, from fellow students, researchers, administrators to technical staff, all of whom I’ve asked for favours now and again or had to listen to my moaning. Thanks also to my colleagues in the SEARCH group at the University of Groningen. Thank you for enjoyable times and interesting discussions.

I’m grateful to all the people that have participated in the various studies that I conducted for this research. Thanks to all students for their willingness to participate in their own time. Thanks to all people at the two organisations (“SoftCom” and “NewCorp”) who were willing to find some time in their busy schedules.

Last but not least, I thank my friends, both in the Netherlands and in Ireland, and my family: my brother Kees, my sister Roelien, and my parents, Ali and Herman, who have supported me during my studies.
Summary

THERE ARE VARIOUS scenarios for a software development organisation to adopt Open Source Software (OSS). One scenario is the adoption of OSS products in order to integrate them as components into a final software product. Another scenario is the adoption of OSS development practices within the confines of an organisation; this is called “Inner Source”. In both cases, the software product that is developed is used in component-based software development (CBSD) to build large-scale products. Also, in both cases, such software is developed by a “community”, or using Raymond’s metaphor, in a “bazaar”. In the first case, the bazaar is external, and in the second case the bazaar is internal to the organisation.

Both scenarios have been reported as providing benefit to organisations. However, little research has been conducted to identify and document the challenges that practitioners may encounter in these two scenarios. The research in this thesis is divided into two strands, one for each scenario.

The first research strand consists of a systematic identification of challenges in using OSS in product development. One of the most frequently reported challenges is the evaluation and selection of components. This was followed by an analysis of existing OSS evaluation methods proposed so far. These methods seem to pay little attention to a product’s software architecture, which has been identified as a critical success factor in CBSD. To gain insight whether or not practitioners are, in fact, interested in architectural knowledge (AK) of an OSS product, an empirical survey was conducted. One of the findings was that practitioners are interested in certain cases in a component’s architectural patterns. Since there exists no guidance to identify architectural patterns (as opposed to design patterns), the thesis proposes a process to guide the task of identifying architectural patterns in an OSS product.

The research continued by investigating the second scenario, which is Inner Source. This strand of research started with an investigation of challenges at an Inner Source organisation. These challenges were categorised and compared to the challenges identified for the external bazaar. This was followed by a literature-based framework that an organisation may use to assess its fit with Inner Source.

In summary, by identifying and documenting challenges and needs, and by proposing guidance to practitioners, the contributions of this thesis support product development with software from the bazaar.
Note

The author hereby declares that, except where duly acknowledged, this thesis is entirely his own work. This point is particularly emphasised in relation to all figures and tables. Where these have been reproduced, the original sources are acknowledged. Otherwise, they represent the author’s own work.
1.1 INTRODUCTION

The Open Source Software (OSS) phenomenon is a topic that has actively been researched for the last 15 years or so. Though the term “Open Source” was only coined in 1998 (Raymond, 2001, p.175), the phenomenon has been around for much longer (mostly by the name “Free Software”), since the early 1960s. Researchers have investigated a large variety of aspects of the topic, focusing both inwards (e.g., developer motivation and project characteristics such as size and growth), and focusing outwards (e.g., adoption) (Fitzgerald, 2006).

Raymond’s seminal work “The Cathedral and the Bazaar” (Raymond, 2001) is among the first with such an inwards focus, and presents an in-depth description and analysis of the OSS phenomenon. Raymond uses metaphors to characterise the difference between traditional software development projects (the “cathedral”, referring to the top-down planning approach comparable to the way cathedrals were designed and built in the Middle Ages) and OSS development projects (the “bazaar”, referring to the “open market” of individuals contributing in a seemingly uncoordinated way to the software). While topics focusing inwards on OSS are of interest to gain an understanding of the phenomenon itself, an outward focus is of interest to understand the implications for the software landscape. There is a large number of high-quality OSS products available that may be viable alternatives to commercial software development (Fitzgerald, 2006). One question that

---

1 The first edition of the book was published in October 1999.
is of interest to industry is: how can organisations benefit from adopting OSS?

The OSS research community has studied OSS adoption in industry extensively. In recent years, a number of OSS adoption scenarios for software development organisations have been identified (Hauge et al., 2010); these are discussed in detail in Chapter 2. This thesis focuses on two of these scenarios in particular, which are briefly outlined here.

The first adoption scenario that is relevant to the research in this thesis is the use of OSS components in Component-Based Software Development (CBSD). OSS products can be used as an alternative to Commercial Off-The-Shelf (COTS) products (Hissam and Weinstock, 2001). CBSD has become a standard approach to build large-scale software systems (Szyperski, 2002). An important issue in CBSD is the selection of suitable components; it is essential to properly evaluate a component in order to prevent problems at a later stage in the integration and development phase. However, a variety of problems may arise in CBSD, many of which can be attributed to “architectural mismatch” (Garlan et al., 1995). Software Architecture (SA) has been recognised as an important success factor in constructing large-scale software systems, including in CBSD (Bosch and Stafford, 2002), and has become an active sub-discipline within the software engineering research arena (Shaw and Garlan, 1996). Though SA is a research area involving a variety of topics, in the context of architectural mismatch, availability of architectural knowledge (AK) (such as design decisions and architectural patterns) has become an actively studied topic. Since 2006, there has been a dedicated workshop to Sharing and Reusing Architectural Knowledge (SHARK) (Lago and Avgeriou, 2006). Long before the emergence of AK as an important topic within the software architecture research community, one particular type of AK, architectural patterns or styles, had already received much attention from both researchers and practitioners (Buschmann et al., 1996; Shaw, 1995b). A pattern is defined as a common solution to a recurring design problem in a certain context (Buschmann et al., 1996). Architectural patterns have been shown to be
an effective means to evaluate software architectures as it provides insights into a product’s quality attributes (e.g., performance and reliability) (Harrison and Avgeriou, 2011), and may therefore be a promising approach to support the evaluation of OSS components in CBSD. This is discussed in further detail later in this chapter as well as in Chapter 4.

The second OSS adoption scenario that is relevant to the research presented in this thesis is the adoption of OSS development practices, a phenomenon that is also known as Inner Source (Wesselius, 2008). In this scenario, a software development organisation develops software in-house in a way that mimics the way in which OSS is developed. While there is no definite set of OSS development practices that is shared by all OSS projects, common practices include universal access to all project artefacts, release early and often, and “community” peer-review (rather than assigned reviews to particular members of the development team or planned review meetings).

1.2 RESEARCH OBJECTIVE

There are two common characteristics of the two above-mentioned adoption scenarios. Firstly, the software product is developed by a community. In the first scenario, the community is external to the organisation (Open Source), whereas in the second scenario, the community (or “bazaar”) thrives within the confines of an organisation (Inner Source). Thus, Open Source and Inner Source are two flavours of a “bazaar” development environment. The second common characteristic is that in both cases, the developed software is eventually integrated into a final product.

Product development with software from a “bazaar” is a topic that has not been studied extensively. Though there are a number of case studies and lessons-learnt reported on using OSS in product development (e.g., (Ruffin and Ebert, 2004; Jaaksi, 2007)), these are individual reports, and do not provide a complete view of issues that an organisation needs
to consider. Furthermore, the topic of Inner Source has received little attention, and is mostly limited to a handful of case studies (e.g., (Wesselius, 2008; Dinkelacker et al., 2002; Gurbani et al., 2006)). It is not clear what challenges may arise in such a scenario, nor is there much insight into when Inner Source can thrive within an organisation.

This research was therefore concerned with the following two research objectives:

1. To document challenges and to provide guidance in product development with OSS components.
2. To identify challenges in Inner Source and to provide insights into when and how Inner Source can thrive.

1.3 RESEARCH OVERVIEW AND CONTRIBUTIONS

In order to achieve the objectives of this research, the researcher has conducted a number of studies (numbered S1 to S6). To address the first objective, the following four studies were conducted:

S1. A Systematic Mapping Study (Kitchenham and Charters, 2007) to identify, aggregate and categorise challenges that may arise in product development with OSS products.

S2. A literature study to identify and compare OSS evaluation methods that have been proposed so far in the literature. This study includes the development of a comparison framework (grounded in literature) that is used to compare a number of the OSS evaluation methods.

S3. An exploratory survey to empirically investigate the importance of architectural knowledge of OSS products.

S4. An empirical study (based on the Improvement Paradigm documented in (Parra et al., 1997)) to develop and assess a process to guide practitioners to identify architectural patterns in OSS products.

In order to address the second research objective, the researcher conducted the following two studies:
S5. An **industrial case study** to identify, document and categorise challenges in an Inner Source organisation. The identified challenges are also compared to the challenges related to the use of OSS products (see S1).

S6. A **literature review** to develop a framework to guide the assessment of an organisation’s fit with Inner Source. The framework-based assessment is demonstrated with data drawn from an in-depth **industrial case study** at an organisation that had indicated an interest in adopting Inner Source.

The studies listed above were all conducted in an incremental research process design. Figure 1-1 shows the “process view” of the research design; the identifiers S1 to S6 in the figure correspond to the studies presented in this thesis and are listed above. Prior to conducting these studies, the researcher had conducted a systematic mapping study of empirical research in OSS. This study (numbered S0, as it is not presented in this thesis) provided the researcher with initial insights into the OSS research arena. Most of the studies have also been published in one or more papers (numbered P1 to P8), as shown in the figure.
Figure 1-1. Overview of studies and publications.

Details of the papers (numbered P1 to P8 in the figure above) are as follows:


All papers were primarily authored by the researcher. The various co-authors contributed useful feedback regarding the contents and structure of the papers. The co-authors have also edited text, and in some cases contributed small parts of text, but the majority of the text was written by the researcher. Furthermore, spelling and grammatical issues were also contributed as well as suggestions to improve the presentation of diagrams and figures. Section 1.4 discusses in which chapters the results presented in the published papers are presented.

These studies have resulted in six key contributions of this thesis. The contributions can be categorised along two dimensions: (1) the bazaar type (Open Source or Inner Source), and (2) the type of contribution (“challenges & needs” or “solutions”). This is shown graphically in Figure 1-2.

![Figure 1-2. Contributions of this thesis, categorised along dimensions of bazaar type and contribution type.](image-url)
The figure shows that the research has investigated both flavours of “bazaars” (Open Source and Inner Source). The contributions can be categorised as “challenges & needs” or “solutions”. Contributions that aggregate and analyse existing literature are classified as “challenges & needs”. In the context of OSS, these are the aggregation of challenges related to product development with OSS components, a framework-based comparison of OSS evaluation methods, and an empirical investigation of the need for, and importance of, architectural knowledge of OSS products. Furthermore, some of the empirical studies conducted for this research are also of a documenting nature. Such contributions may provide useful to both practitioners and researchers. For practitioners, overviews and classifications are presented that reflect the current “state” of the research area. For researchers, such overviews may provide a starting point and motivation for further research.

Two contributions are categorised as “solutions” to practitioners. The guidance for practitioners using software from OSS bazaars is a process to identify architectural patterns in OSS products. The guidance provided in Inner Source bazaars is a framework to assess an organisation’s fit with adopting Inner Source; that is, a means to assess how well Inner Source would fit within an organisation which can help in the decision whether or not to adopt Inner Source.

1.4 STRUCTURE OF THE THESIS

The remainder of the thesis is structured as outlined below. Chapters 2 to 4 present relevant background information that introduces the context of the research, followed by Chapter 5 which provides an overview of the research design. This is followed by Chapters 6 to 11 which present the results of the six studies (S1 to S6). Figure 1-3 shows how the various background and results chapters relate to one another. The figure also considers elements from Figure 1-1 and Figure 1-2, namely, references to the conducted studies (S1 to S6), published papers (P3 to P8) and contributions (C1 to C6). Please note that Chapter 12 (Summary and Future Work) is left out in this figure.
Figure 1-3. Structure of this thesis.

Chapter 2 reviews the literature on the emergence of OSS adoption in industry. The chapter reviews the different OSS adoption scenarios, and draws parallels between the two scenarios mentioned above, namely, the use of OSS products in CBSD on the one hand and Inner Source on the other. The chapter also provides an overview of OSS development practices.

Chapter 3 focuses more closely on Inner Source. The chapter presents the two Inner Source models that have emerged, namely, project-based Inner Source and infrastructure-based Inner Source. Furthermore, the chapter also reviews the literature that provides guidance and suggestions on how to adopt Inner Source.

Chapter 4 addresses the role of software architecture in using software from the “bazaar”. As outlined above, software architecture plays an important role in CBSD. The chapter presents a brief introduction to the field of software architecture, and introduces relevant concepts for the research conducted, such as architectural knowledge.
Chapter 5 presents an overview of the research process that the researcher followed. In particular, the research objectives are revised, and six research questions are derived. The chapter then provides an overview and justification of the research methods used to conduct the studies.

Chapters 6 to 11 address each of the six research questions that are derived in Chapter 5; these six questions are answered through studies S1 to S6 mentioned above. Chapters 6 to 9 focus on OSS, whereas Chapters 10 and 11 focus on Inner Source.

Chapter 6 presents the findings of a systematic mapping study to identify and categorise challenges related to product development with OSS components, as they have been reported in the literature. Twenty-one challenges have been identified in six categories. One of the most reported challenges is the evaluation and selection of appropriate components.

Chapter 7 presents the findings of a literature review, through which the researcher systematically identified 20 OSS evaluation frameworks, approaches and methods that have been reported in the literature. The chapter proceeds with the development of a comparison framework (grounded in the literature) that is subsequently used to compare six of these OSS evaluation methods. All methods propose a set of evaluation criteria, some of which also consider architectural aspects, and in particular quality attributes (so-called “ilities”), such as performance and reliability. Such type of information is considered one type of “architectural knowledge” (AK). The OSS literature, however, has paid little attention so far to the importance of software architecture or architectural knowledge in OSS evaluation.

Chapter 8 presents the findings of a small-scale exploratory survey to investigate the importance of architectural knowledge in integrating OSS. The study presents an analysis from data drawn from 12
interviews with practitioners. The chapter provides empirical evidence that practitioners are interested in certain types of architectural knowledge (which are identified and categorised). One particular type of AK that practitioners indicated to be interested in is architectural patterns.

Chapter 9 presents the design and evaluation of a process to identify architectural patterns in OSS products. Following the “Improvement Paradigm” methodology (Parra et al., 1997), the researcher conducted a number of studies to identify steps and data sources, to provide systematic guidance to practitioners that wish to identify patterns in OSS products. As briefly mentioned above, architectural patterns have been shown to be useful in evaluating software architectures (Harrison and Avgeriou, 2011). Therefore, such information can assist practitioners to assess the quality attributes (e.g., performance, reliability) of an OSS product, and may therefore be help in the evaluation of OSS products.

Chapter 10 presents the findings of an in-depth case study investigating challenges in an organisation that has adopted an Inner Source model. The case study identified 13 challenges, 10 of which could be mapped to the challenges identified in the systematic mapping study presented in Chapter 6. The case study also identified a number of approaches that the organisation has adopted to mitigate the challenges encountered.

Chapter 11 presents the development and justification of a framework, grounded in the literature that provides guidance and advice on adopting OSS development practices, which can be used to assess the suitability of an organisation to adopt Inner Source. In order to demonstrate how the framework can be used, an in-depth, industrial case study was conducted at an organisation that had indicated an interest to adopt Inner Source. The framework helped the researcher to reveal a number of “tension points”, or incompatibilities that might hinder the adoption of Inner Source. These tension points should be considered if the organisation wishes to proceed to adopt Inner Source.
Chapter 12 summarises and concludes this thesis. The chapter provides a brief overview of the contributions made in this thesis (in particular by reviewing the results presented in Chapters 6 to 11). The chapter also discusses implications of these findings for future research as well as for practice.
Chapter 2

The Emergence of Open Source Software

2.1 INTRODUCTION AND CHAPTER LAYOUT

This thesis focuses on integration of software products developed using Open Source Software (OSS) development practices. This may be either “real” OSS developed by an OSS community of (typically) volunteers, or software developed within a corporate environment using OSS development (OSSD) practices. This phenomenon is known as “Inner Source”, though different terms have been used in the literature. The purpose of this chapter is to set the context of the research presented in this thesis by reviewing the relevant literature on the emergence of Open Source Software (OSS) in industry.

Firstly, an overview is provided of the different scenarios for adopting Open Source Software in industry (Section 2.2). Then, Section 2.3 draws a comparison between the integration of OSS products and Inner Source Software (ISS) products. Section 2.4 presents a discussion of OSS development practices. Section 2.5 concludes the chapter.

2.2 ORGANISATIONAL ADOPTION OF OPEN SOURCE SOFTWARE

Over the last decade or so, organisations have increasingly adopted Open Source Software (Hauge et al., 2010). Adoption of OSS can be done in a number of ways. Various researchers have identified a number of different scenarios; this section provides an overview of these scenarios.

Theunissen et al. (2006) identified four levels of increasing involvement in OSS projects (sorted from lowest level to highest level of involvement):
• Simply using a product;
• Modifying a product without sharing the modifications;
• Modifying a product and contributing the changes back into the community;
• Initiate and/or manage an OSS project;

In later work, Theunissen et al. (2008), also mentioned the adoption of “in-house OSS development style”, but did not discuss this option in further detail. Other researchers have listed similar scenarios to benefit from OSS as well. Van der Linden (2009) lists the following:

• Using OSS development tools;
• Using OSS components in products;
• Applying OSS development practices;
• Opening up products (as OSS);
• Establishing a symbiotic relationship.

In a recent systematic review of the literature on OSS adoption, Hauge et al. (2010) identified the same five options as Van der Linden. Hauge et al. note that these five options are all adoption options in the context of software development, and identified a sixth option in a more general (non-software development) context, namely, the deployment of OSS products. An example of this is the adoption of productivity suites such as OpenOffice.org at local governments (e.g., (Ven et al., 2006)). Another systematic review by Höst and Oručević-Alagić (2011) focused on the adoption of OSS in commercial software development, and also identified options two to five, listed above as identified by Van der Linden.

Figure 2-1 presents a graphical mapping of how the different scenarios identified by different research groups relate to one another. Equal adoption scenarios that are identified by different groups are visually grouped in grey-coloured boxes. The following sub-sections briefly discuss the five scenarios that are identified by more than one research group.
Figure 2-1. Different ways for organisations to benefit or adopt from OSS. Arrows indicate equivalence or similarity. Grey boxes group equivalent options.

2.2.1 Adoption of OSS development tools

The first option is the adoption of OSS development tools for software development. A few examples of OSS development tools are: the Eclipse IDE (Integrated Development Environment), GCC (GNU Compiler Collection), and Trac (wiki and issue tracking system). An extensive overview of various OSS tools is presented in (Koranne, 2010). While organisations often indicate OSS licenses to be one obstacle to adopt OSS, this does not apply for the adoption of OSS tools, since the tools themselves do not usually become part of the final product.

Adoption of OSS development tools was identified as one way of adopting OSS by both Van der Linden (2009) and Hauge et al. (2010). It can also be considered as one type of “using an OSS product”, as identified by Theunissen et al. (2006).
2.2.2 Integrating OSS components in product development

The second option to adopt OSS is the integration of OSS component into products (Van der Linden, 2009; Hauge et al., 2010; Höst and Oručević-Alagić, 2011). The use of OSS components is an alternative to using Commercial-Off-The-Shelf (COTS) components to build large-scale software systems (Hissam and Weinstock, 2001). Over the past decade or so, OSS products have become more mature and viable for use in commercial products (Fitzgerald, 2006). The integration of OSS components into a product can also be considered as one type of “using an OSS product”, as suggested by Theunissen et al. (2006), though they did not specify this specific usage explicitly.

2.2.3 Using OSS development practices

The third option is the adoption of OSS development practices. Van der Linden (2009) calls this “Inner Source”, but other terms are in use, such as “Corporate Open Source” (COS) (Gurbani et al., 2010) and “Progressive Open Source” (POS) (Dinkelacker et al., 2002). Rather than adopting an OSS product, an organisation adopts OSS development practices within its confines for its in-house software development. Chapter 3 discusses the Inner Source phenomenon in more detail.

2.2.4 Opensourcing a product

Another option for an organisation to be involved in OSS is opening up a product. That is, the organisation gives away the source code of an in-house developed (or an acquired) product and makes it available as an OSS project. Ågerfalk and Fitzgerald (2008) named this phenomenon “opensourcing”. This option is comparable to level four (initiating and/or managing an OSS project) listed by Theunissen et al. (2006).

While at first sight this seems as giving away intellectual property, this is a strategic decision that an organisation can make, in particular for commodity software. Commodity software is software that does not contain expensive intellectual property (Van der Linden, 2009; Van der Linden et al., 2009). An organisation may enjoy several benefits from this
strategy, as it enables faster development and maintenance due to a larger developers pool (Van der Linden, 2009; Raymond, 2001). Raymond discusses this strategy as one of the possible business-models for OSS and calls this “Widget Frosting” (Raymond, 2001, p.135). Hauge et al. (2010) listed a number of organisations that have been opened up their products, such as JBoss, MySQL and Qt Software. These organisations control the development of the products and interact with the community that gathered around them. Additionally, these organisations have built a business model around this by selling support. For instance, JBoss provides a “community” version of the JBoss application server\(^2\), as well as an “enterprise” version\(^3\). Raymond refers to this business model as “Give Away the Recipe, Open a Restaurant” (Raymond, 2001, p.136).

(Note that this business model is also used by organisations, such as Red Hat, which sell support on a product (Linux) that was already open source.)

### 2.2.5 Participating in OSS development

The fifth option is what Van der Linden calls “establishing a symbiotic relationship with an OSS community”. Hauge et al. (2010) refer to this as participation in the development of an OSS product. Höst and Oručević-Alagić call this category “Business models with OSS”. Van der Linden (2009) gives an example of Philips Healthcare, an organisation that is actively participating in the Subversion community. Through this participation, Philips Healthcare has developed and donated a tool for resolving merge conflicts. This option is similar to the third level of involvement listed by Theunissen et al. (2006).

### 2.3 COMPARISON OF OPEN SOURCE SOFTWARE AND INNER SOURCE SOFTWARE INTEGRATION

This research is focussed on supporting developers in building products with software “from the bazaar”, whether that is an “open” bazaar (Open

---

\(^2\) [http://www.jboss.org](http://www.jboss.org)  
\(^3\) [http://www.jboss.com](http://www.jboss.com)
Source) or an “internal” bazaar (Inner Source). Of the different adoption options discussed above, the researcher argues that two options of OSS adoption are directly relevant to this research focus. The first, obviously related, option is “OSS in component-based software engineering”. The second is, perhaps not as obviously related, the use of OSS development practices (“Inner Source”). In Inner Source, a software development organisation has adopted OSS development practices to build software components, which can be subsequently integrated into products. Since the software components that are integrated into products are developed following similar development practices, the researcher argues that the scenarios of integrating OSS and integrating the “Inner Source Software” are comparable. This is illustrated in Figure 2-2.

The diagram on the left-hand side in Figure 2-2 shows two software-developing organisations (A and B) that integrate an OSS product. The OSS is developed by contributors dispersed all over the world, who communicate through the Internet (e.g., through mailing lists, defect trackers and IRC channels). In other words, the OSS community is
external to the organisation. Sometimes, organisations that integrate OSS products can become active members of such a community by contributing (e.g., bug reports and feedback, feature requests and source code) or sponsoring.

The diagram on the right-hand side in Figure 2-2 shows a software-developing organisation with two divisions (A and B) that integrate Inner Source Software (ISS). The software is developed within an organisation that has adopted Inner Source, and therefore uses certain development practices common in OSSD. In other words, the organisation integrates software that is developed using similar practices as used in OSSD, and has effectively an internal “community”, or “market place” (Wesselius, 2008). Wesselius calls this “creating the bazaar within the cathedral” (Wesselius, 2008), using Raymond’s metaphors (Raymond, 2001). In both OSSD and ISSD, developers may choose voluntarily to contribute to the ISS product (the shared asset).

### 2.4 OPEN SOURCE DEVELOPMENT PRACTICES

So far, this chapter has focused on the different ways in which organisations may benefit from OSS. Section 2.3 discussed integration of both OSS components and ISS components, after which it was argued there are strong similarities between the two integration scenarios. In order to gain a further understanding of how these components are developed, it is imperative to discuss what constitutes OSS development practices.

Open Source Software Development (OSSD) is often characterised as software developed by geographically distributed volunteers, without formal work assignments and project planning, and rapid release-and-fix development cycles. Together, these developers form an OSS project’s community. However, as Østerlie and Jaccheri (2007) argue, there is no empirical evidence that supports the assertion that OSSD is such a homogeneous phenomenon. Feller and Fitzgerald (2002) argue that there is no single OSSD process, and note that just a
handful of projects (e.g., Mozilla Firefox, Apache, Linux kernel) keep recurring in OSSD research. Similar findings were reported in (Stol et al., 2009) and (Crowston et al., In Press). Each OSS project may follow a different process, but all OSS projects share a common philosophy (Theunissen et al., 2008). It is not the goal of this section to describe in definite terms what OSS development is, but rather to inform the reader of what is typically meant by the term OSS development.

Before discussing common OSS development practices, though, it is important to understand who performs these practices. Therefore, Subsection 2.4.1 discusses the structure and characteristics of a typical OSS development community, after which common OSS practices are discussed in 2.4.2.

2.4.1 The Open Source Software development community

OSS is commonly written by voluntary developers, who together form a project’s community. This section discusses some of the common characteristics and structure of OSS communities.

Since 1997, Larry Wall, creator of the Perl programming language (a well known OSS project), has presented his annual State of the Onion talk. In his second4 “State of the Onion” speech in 1998, Wall said5:

“If you cut an onion, it looks like this:

---

4 The first State of the Onion speech was only named “State of the Onion speech” in retrospect.
5 The full transcript of the speech is online at: http://www.wall.org/~larry/onion/onion.html; the onion metaphor is also mentioned in Wall, L. (1999) Diligence, Patience, and Humility, in: DiBona, C., Ockman, S. & Stone, M. (Eds.) Open Sources: Voices from the Open Source Revolution. Sebastopol, California, O'Reilly
If we take this to be a picture of the world of Perl, then I must be that little bit of onion inside. Around me are some of the early adopters of Perl, who are now revered as heroes of the revolution. As more people have joined the movement, new layers have been added.”

Wall was the first person to compare the structure of the Perl community to an onion, and placed himself in the middle (core) of the onion. Researchers have since studied OSS communities in great detail, and described OSS communities using the same onion metaphor⁶. Crowston et al. (2004) presented the onion model shown in Figure 2-3.

![Figure 2-3. The onion model representing the community structure of a typical OSS project (taken from Crowston et al. 2004).](image)

It is important to note that the onion model is a theoretical model of the structure of a typical OSS project. This implies that by far not all OSS projects conform completely to this model, since each OSS project is unique in its structure, governance, and its level of success in attracting contributors.

According to the onion model, there is a relatively small team of core developers. No general statements can be made about the size of this core, as this varies widely among OSS projects. However, based on a case

---

⁶ It is not clear whether these researchers were aware of Wall’s first use of the onion metaphor.
study of the Apache web server, Mockus et al. (2002) hypothesised that the core team has an upper limit of 10 to 15 members, due to communication and coordination overhead issues. The core team members provide most of the code, and take responsibility for overseeing the design and evolution of the project. In a second case study, Mockus et al. studied the Mozilla web browser, and found that the size of the core teams (one team for each main module) was significantly larger (22 to 36 members).

Around this core team, there is a group of co-developers that is an order of magnitude larger than the core team. Co-developers review and contribute code, as well as bug fixes.

The next layer of the onion surrounding the layer of co-developers represents a group of active users, who are not developers. Active users closely follow a project’s progress by downloading and using the latest releases, reporting bugs and suggesting new features.

The outer layer of the onion represents passive users, who merely use a product, but do not contribute anything to the OSS project. Since it is practically impossible to define the size of this group (since they do not interact with the community and do not make themselves known), Figure 2-3 shows this layer with a “spiked” (undefined) outer border.

Within the onion-shaped community, members may transition from one layer to another (Gacek and Arief, 2004). Passive users may become more engaged with the OSS project as their interest grows, and start submitting bug reports or feature requests, which makes them “active users”. Non-developing active users may start contributing code, either to fix a bug or to implement a new feature; this transitions them from being non-developers to co-developers. After submitting a certain number of high-quality code contributions, co-developers may “earn” write-access (for instance, by being nominated by other senior members) and become “trusted” members of the development community. This allows them to commit directly to the central repository, rather than submitting contributions as patch files. Co-developers may become
increasingly involved in the development of one or more modules of the OSS project, and may become a member of the core team.

### 2.4.2 Open Source Software development practices

As stated before, there is no single OSSD process; each OSS project has its own unique development process. Mockus et al. (2002) list four main differences from “the usual industrial style of development”, which they found are most often mentioned:

- OSS systems are built by potentially large numbers of volunteers;
- Work is not assigned, but people undertake the work they choose to undertake;
- There is no explicit system-level design, or even detailed design (citing (Vixie, 1999));
- There is no project plan, schedule or list of deliverables.

Note that the last three differences are all negative assertions, in that artifacts do *not exist* or that activities are *not performed*. Feller and Fitzgerald (2002) list the following generic characteristics of OSS development processes (stated as positive assertions):

- Parallel (rather than linear) processes (development, debugging);
- Large community of globally distributed developers;
- Truly independent peer review;
- Prompt feedback to user and developer contributions;
- Participation of highly talented and motivated developers;
- Increased levels of user involvement;
- Extremely rapid release schedules.

Various authors have addressed the question of what OSSD practices *are*, and have attempted to provide an overview of the key practices that are commonly used. Rather than adopting one set of practices as identified by one research group to structure this presentation, the researcher decided to draw from different sources. This sub-section provides an overview of typical practices followed by OSS projects. The remainder of this sub-section discusses these practices in more detail.

#### 2.4.2.1 Planning, requirements analysis and feature requests

A key difference between “traditional” software development life cycles and OSS development is requirement analysis (Scacchi, 2004). In OSSD,
requirements are typically not formally elicited, captured and analysed (Scacchi, 2004). Instead, requirements are typically “asserted after the fact”. Feller and Fitzgerald (2002) argued that:

planning, analysis and design are largely conducted by the initial project founder, and are not part of the general OSS development life cycle.

This, of course, depends on the project’s starting point; a project may start as an open source project from the beginning, or start from an existing commercial or research system that is “opened up” (Gacek and Arief, 2004). In a recent systematic review on OSS development, Crowston et al. report that OSS projects do not typically engage in formal planning (Crowston et al., In Press). Most projects, though, do keep some sort of development agenda in the form of a TODO list, roadmaps or feature request lists. Most issue trackers, such as Trac7, could also be used for this purpose. Features can be proposed by anyone, including non-developers (e.g., “active users”) and core developers.

2.4.2.2 Feature implementation and defect fixing
The key activity in OSS development is feature implementation. Features can be implemented by both members of the core development team, or by co-developers (Gacek and Arief, 2004). In a case study of the Apache web server, Mockus et al. (2002) found that the top 15 developers contributed more than 88% of the lines of code. On the other hand, it is not uncommon for co-developers to implement new features as well. A second code development activity, besides implementing new functionality, is fixing defects in existing functionality.

2.4.2.3 Code review and inspections
A key characteristic and strength of OSS development is “truly independent peer review” (Feller and Fitzgerald, 2002). Successful OSS projects, which may have attracted tens, hundreds or even thousands of community members can greatly benefit from the large-scale scrutiny of contributions that are submitted by other members. Contributions are

7 http://trac.edgewall.org/
typically patches containing source code, but documentation, tests and bug reports can also be subjected to peer review (Gacek and Arief, 2004). Raymond describes how the Linux kernel project greatly benefited from a large pool of developers spending time on debugging and development, and observed that:

> Given a large enough beta-tester and co-developer base, almost every problem will be characterised quickly and the fix obvious to someone. (Raymond, 2001, p.30).

Raymond has coined this Linus’s Law: “*Given enough eye balls, all bugs are shallow*” (Raymond, 2001).

Peer review can happen two ways: pre-commit and post-commit. _Pre-commit_ reviews are performed before the changes are made (committed) to a central repository (usually a version control system) that contains all project artefacts. This is typically the case with contributions of members that do not have write-access to the repository. Instead of speaking of “write-access”, OSS developers sometimes speak of having a “commit bit”, which means that a developer may directly commit changes to the repository. Contributions are usually sent as a patch file (created using the “diff” utility program, which calculates the differences between two text files), to a mailing list or issue tracker. Any member of the OSS community is then able to “apply” the patch file to try out the changes submitted. After a senior member (with a “commit bit”) has reviewed and approved the contribution, he\(^8\) would apply the patch to the main development branch in the repository (which is often referred to as “trunk”). In other cases, if the contribution is not considered to be good enough, the contributor is asked to make small changes or take a different approach and resubmit.

_Post-commit_ reviews are typically performed on contributions by members that have a commit bit. These members are generally trusted to make good-quality contributions. After the developer has ensured that a contribution (e.g., a bug fix) works on his own local copy of the software, \(^8\)“he” should be read as “he or she” in the remainder of this thesis.
he can commit the changes directly to trunk. Other members are typically informed of the changes, for instance through an automatic email sent out after the change has been committed, or an automated message on the IRC\textsuperscript{9} channel (by an “IRC bot”), which is a common means of communication in OSS projects. Some tools provide a web-based interface that displays the patch (e.g., source code lines added and deleted in green and red, respectively). Such a notification typically contains the name of the committer, a summary of the changes, and (depending on the tools that are used) a link to a web-based interface that displays the patch file. Once informed, other developers can then inspect the changes that were made. In some cases, the committer is asked to revert the changes, if it was found that changes are not a good solution to the problem that is being solved (i.e., implementation of functionality or a bug).

2.4.2.4 Release management

Another key characteristic of successful OSS projects is that they “release early, [and] release often” (Raymond, 2001). As Crowston et al. point out, there is no single OSS approach to perform releases (Crowston \textit{et al}., In Press). Erenkrantz (2003) studied release practices of three projects: Linux kernel, Subversion, and Apache HTTP server. The three projects are compared based on a proposed taxonomy containing six characteristics:

- Release authority: who performs the release?
- Versioning: how are released versions labelled or numbered?
- Pre-release testing: what are the criteria to satisfy prior to a release?
- Approval of releases: what is the procedure for approving releases?
- Distribution: how is the release distributed?
- Formats: how is the release packaged?

Erenkrantz found that release practices differ significantly among the three studied projects. It is therefore difficult to describe release practices in general, however, the job of release manager is typically rotated among a number of experienced developers.

\textsuperscript{9} IRC: Internet Relay Chat
2.4.2.5 Maintenance and evolution

OSS development differs from “traditional” commercial software development with respect to the maintenance phase. Whereas commercial software is at some point released to a customer, after which the maintenance phase starts (usually with a maintenance support contract), OSS projects are in a constant state of flux. As Merilinna and Matinlassi phrased it: “OSS is a moving target – or a dead corpse” (Merilinna and Matinlassi, 2006).

Scacchi (2004) argues that the nature of the maintenance phase of OSS differs from commercial software. He characterises the evolutionary dynamic of OSS projects as reinvention. A good example of this is the Perl programming language, which was first released in 1987. In the year 2000, it was recognised that Perl’s implementation had become too difficult to maintain, as described by Chip Salzenberg10, who previously had made a first attempt in 1998 to re-implement the Perl language in C++, a project called “Topaz”:

Perl's guts are, well, complicated. Nat Torkington described them well. I believe he said that they are "an interconnected mass of livers and pancreas and lungs and little sharp pointy things and the occasional exploding kidney." It really is hard to maintain Perl 5. Considering how many people have had their hands in it; it's not surprising that this is the situation. And you really need indoctrination in all the mysteries and magic structures and so on—before you can really hope to make significant changes to the Perl core without breaking more things than you're adding.

Though Topaz did not succeed as the next implementation of Perl, the need for a new approach was evident throughout the Perl community11. At some point, the Perl community as a whole (rather than an individual’s initiative such as by Salzenberg) decided to re-implement the language. This was also taken as an opportunity to clean up the language syntax itself. The new design is known as Perl 6, and its specification was separated from the implementation.

11 This decision was preceded by a somewhat dramatic “mug throwing” incident, which is described in: http://oreilly.com/catalog/perl6es/chapter/ch01.pdf
A second related example is Parrot\textsuperscript{12}, which is a new runtime engine to run Perl 6 code. Parrot is a spin-off project from the Perl 6 initiative, which was first released in 2001. Parrot version 1.0 was released in March 2009, but even after this milestone it has been in a constant state of reinvention. For instance, Parrot developers have re-implemented Parrot’s garbage collector to address performance issues.

### 2.4.3 The Open Source paradigm as an agile method

OSS development practices have previously been compared to agile development methods (Warsta and Abrahamsson, 2003). Agile software development practices (e.g., Extreme Programming (XP) and Scrum) explicitly accept the fact that software development projects cannot be completely planned; rather, agile methods embrace the fact that projects are always subject to change (e.g., changing requirements). Warsta and Abrahamsson (2003) identified a number of shared characteristics in agile software development methods:

1. Incremental (rapid release cycles, “release early and often”);
2. Cooperative (close cooperation between customer and developers);
3. Straightforward (method is easy to learn and modify);
4. Adaptive (possible to make last moment changes).

Based on their analysis of the OSS development paradigm, they concluded that it complies with all four characteristics listed above. Warsta and Abrahamsson also identified a few key differences:

1. In OSS, requirements are commonly owned, continually evolving and never “finalised”; in a (commercial) agile development the project has a certain end-date;
2. In OSS, development teams are typically large; agile encourages small teams;
3. In OSS, developers are typically geographically dispersed; agile methods encourage collocation of developers;
4. In OSS, the primary objective is to solve a challenging problem, whereas in agile software development it is to deliver value rapidly.

\textsuperscript{12} www.parrot.org
The similarities suggest that the OSS development paradigm offers a number of sound development practices that can be useful in commercial software development. Some of the differences listed above seem to contrast with OSS, e.g., large v. small teams and geographically dispersed v. collocated developers.

2.5 CHAPTER SUMMARY

This chapter has reviewed the literature on the emergence of Open Source Software in industry. In particular, different approaches to adopt OSS that have been identified by different research groups were presented and compared, in order to provide a wide overview of OSS adoption.

This thesis focuses on the integration of software components from the “bazaar”, either in “open air” or within a “cathedral”. This is related to two scenarios of OSS adoption: (1) the use of OSS components in CBSD, and (2) the adoption of OSS development practices within a software-developing organisation, which is referred to as “Inner Source”. In this chapter, the researcher has argued that the integration of OSS components can be compared to the integration of software components that are developed in an Inner Source environment, thereby linking the two scenarios of OSS adoption mentioned above that are relevant to the research presented in this thesis.

The development method by which a software product is developed has a direct influence on the product. Therefore, This chapter continued by presenting an overview of typical OSS development practices, while also emphasising that there is no single, well-defined set of OSS development practices. Rather, OSSD practices are heterogeneous among OSS projects.

The adoption of OSSD practices within corporate settings was briefly addressed. Whereas this chapter has provided an overview of OSSD practices, Chapter 3 which follows next discusses the emergence of “Inner Source”; that is, the adoption of OSSD practices in a corporate setting.
Chapter 3

The Emergence of Bazaar-Style Software Development in Cathedrals

3.1 INTRODUCTION AND CHAPTER LAYOUT

The previous chapter discussed different scenarios for organisations to adopt OSS, one of which is the adoption of OSS Development (OSSD) practices. This phenomenon has been named “Inner Source”, although different authors have used different terms. This chapter focuses on the Inner Source phenomenon in more detail. The chapter begins with a brief discussion of organisations’ motivation to adopt Inner Source (Section 3.2). The chapter continues with a discussion of the two Inner Source models that have emerged (Section 3.3). Characteristics of both models are discussed and compared. This is followed by an overview of reported cases of Inner Source adoption in industry (Section 3.4). This section also addresses the terminology used by different authors. The next section (3.5) provides an overview of guidelines that have been proposed by other authors that organisations can use to adopt an OSS style of working. Section 3.6 presents an overview of a number of high-level categories into which the various guidelines can be classified. The chapter is concluded in Section 3.7.

3.2 MOTIVATION TO ADOPT INNER SOURCE

OSS development is often characterised with terms such as unstructured, chaotic, and unplanned. This is in contrast with the many efforts that have gone into systematic development methods and guidelines that have been proposed. Why, then, would an organisation want to adopt OSS development practices? Gaughan et al. (2009) studied seven
implementations of Inner Source through a review of the literature and a number of interviews with developers and middle management. They list a number of motivations for implementing Inner Source:

- Global Software Development
- Software reuse
- Overcome problems that OSS is reported to have achieved [sic].
- Maintenance
- Quality
- Many eyeballs
- Community building
- Open discussion
- Modular design
- Greater agility
- Decrease costs

Unfortunately, Gaughan et al. (2009) do not further elaborate on each of these motivations. However, it is the researcher’s interpretation that implementing Inner Source may help to achieve, facilitate, implement, or benefit from the above reasons (e.g., Inner Source helps to achieve software reuse; Inner Source helps to facilitate community building).

Gaughan et al. (2009) also report on the benefits reported after implementation of Inner Source. Achieved benefits that have been reported are:

- Code quality;
- Community debugging;
- Leveraging of Open Source to gain perceived benefits;
- Software reuse;
- Faster development;
- Personnel awareness of companies codebase.

These benefits are not further described; note that some of the benefits are overlapping. For instance, “community debugging” is overlapped by the more generally stated benefit of “leveraging of Open Source to gain perceived benefits”. Summarising, there are a number of reasons for an organisation to consider the adoption of Inner Source.


3.3 INNER SOURCE ADOPTION MODELS

Gurbani et al. (2010) identified two different models to implement an Inner Source program: *infrastructure-based* and *project-based*. These models are discussed next.

3.3.1 Infrastructure-based Inner Source model

In an *infrastructure-based* Inner Source model, an organisation provides the required infrastructure (e.g., mailing lists, collaborative development environments (CDEs) and repositories, development tools) to allow developers host individual software development projects. There have been several reports on what can be considered infrastructure-based Inner Source initiatives. Riehle et al. (2009) discuss this model in more detail and how it was applied at SAP. Lindman et al. (2008) describe the *iSource* initiative at Nokia, which is also an infrastructure-based program. Dinkelacker et al. (2002) named the leveraging of OSS methods and tools within HP Corporation “Progressive Open Source” (POS). POS is the umbrella term used at HP to introduce Open Source concepts in HP’s development practices (Dinkelacker et al., 2002), and it consists of 3 tiers, as shown Figure 3-1.

![Figure 3-1. The 3 tiers of Progressive Open Source (taken from (Dinkelacker et al., 2002)).](image)

The first tier is named “Inner Source”, and “refers to the application of the Open Source approach and benefits to developers within the corporate environment” (Dinkelacker et al., 2002). The second tier is
named “Controlled Source”, and extends Inner Source by giving access to the source code to selected corporate partners. The third tier is “Open Source”, and refers to the usual meaning of OSS projects.

POS has been applied within HP Corporation through two programs: (1) Corporate Source initiative (CSI, POS’ first tier), and (2) Collaborative Development Program (CPD, POS’ second tier). Both CSI and CDP are infrastructure-based programs. The “translation process” of OSSD into practices within HP (within its POS program) and its partners has been further reported in (Melian, 2007; Melian and Mähring, 2008).

3.3.2 Project-based Inner Source model

In the project-based Inner Source model, there is one division (called the “core team”) within the organisation funded by other divisions that take over the responsibility of a critical resource (shared asset) and makes it available to the other divisions as a shared asset (Gurbani et al., 2010). Gurbani et al. have reported on a project-based Inner Source model applied at Alcatel-Lucent (Gurbani et al., 2006; Gurbani et al., 2010), where the shared asset was a Session Initiation Protocol (SIP) software stack. Wesselius reports on a project-based Inner Source model as applied at Philips Healthcare and discusses business model aspects (Wesselius, 2008). At Philips Healthcare the shared asset is a platform for software product lines in the medical domain.

Gurbani et al. (2010) have identified a number of roles within the core team at Alcatel-Lucent. These are listed and described in Table 3-1. The different roles have emerged over time; Alcatel-Lucent’s Inner Source initiative (which they call “Corporate Open Source”) was started several years ago. Note that not all project-based Inner Source initiatives have, or need to have these roles; these are specific to Alcatel-Lucent. However, they convey valuable information, in that the role descriptions suggest various activities and responsibilities, which could be useful to other organisations that wish to adopt a project-based Inner Source initiative.
Table 3-1. Roles in the core team, as identified by Gurbani et al. (2010).

<table>
<thead>
<tr>
<th>Role</th>
<th>Responsibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Liaison</td>
<td>Overall responsibility for the product, manages all activities performed by the core team, and interfaces with each business division for new work requests.</td>
</tr>
<tr>
<td>Chief architect</td>
<td>Works closely with the liaison to review and prioritize the feature list, serves as the advocate for internally generated development requests, and communicates planning information to the rest of the corporation.</td>
</tr>
<tr>
<td>Construction, verification and load bring-up engineers</td>
<td>Interface directly with the business division using the asset to provide support for release management tasks such as compilation, load bring-up, and verification. They also perform other support tasks such as maintaining the common asset’s software and tool development environment, documentation, authoring release notes and web design.</td>
</tr>
<tr>
<td>Project manager</td>
<td>Crucial to assist in release and load planning; to manage the tools used to define and track features; and to ensure process compliance, which are often endemic to the corporate world.</td>
</tr>
<tr>
<td>Core development team</td>
<td>Completion of full cycle development tasks, including architecture, design and unit testing, under the leadership of the liaison and working closely with the chief architect.</td>
</tr>
<tr>
<td>Release advocate</td>
<td>Ensure that the code changes for all features were submitted on time and keeping track of all business division-specific impacts for the particular release.</td>
</tr>
<tr>
<td>Business delivery advocate</td>
<td>Assigned to a business division that intended on using the common asset but were new to the concept of the project-specific inner source model. To assist in the difficult task of build integration to the business division.</td>
</tr>
<tr>
<td>Feature advocate</td>
<td>To see a particular feature to completion. To approve design documents, perform code inspections and ensuring that the change aligns with the overall software architecture.</td>
</tr>
</tbody>
</table>

3.3.3 Comparing infrastructure- and project-based Inner Source

In this section a comparison of the infrastructure- and project-based Inner Source models is presented. Table 3-2 lists a number of key attributes, and how these can be characterised in both infrastructure- and project-based models. Note that these are typical, not necessary characteristics; a description of these characteristics may help to understand the difference between the two models. These characteristics are based on observations from the literature and an empirical study that has been reported in Chapter 10 of this thesis. Finally, each organisation implements Inner Source in its own way, tailoring it to its context and needs (Gaughan et al., 2009). An organisation may organise its Inner Source initiative as a

---

13 The researcher gratefully acknowledges Vijay K. Gurbani for useful advice as to the differences between the two Inner source models (personal communication).
mixed approach, taking characteristics of both infrastructure- and project-based models.

Table 3-2. Comparison of Infrastructure-based and project-based Inner Source models.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Infrastructure-based</th>
<th>Project-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reuse</td>
<td>Opportunistic, ad-hoc. Maximise number of projects to be shared within the organisation</td>
<td>Strategically planned. Optimising reuse of critical assets</td>
</tr>
<tr>
<td>Support</td>
<td>Optional, project-dependent</td>
<td>Essential</td>
</tr>
<tr>
<td>Typical interface</td>
<td>Forge or Portal (e.g., SourceForge.net) and project websites.</td>
<td>Project website and repository (version control system)</td>
</tr>
<tr>
<td>Owner/Maintainer</td>
<td>Individual project creator/owner(s)</td>
<td>Central core team</td>
</tr>
<tr>
<td>Contributors</td>
<td>Anyone interested</td>
<td>Core team and customers</td>
</tr>
<tr>
<td>Type of software</td>
<td>Discrete software packages (e.g., utilities, tools, compilers, shells).</td>
<td>Critical assets (e.g., platform of a software product line). Primary technology, rather than a tool.</td>
</tr>
</tbody>
</table>

The remainder of this sub-section discusses these attributes in more detail.

3.3.3.1 Reuse

Inner Source enables reuse of software, but the way this is done in the two models is different. Reuse in the infrastructure-based model is typically opportunistic and ad-hoc, since the main goal is to maximise the number of projects to be shared within the organisation. Project-based Inner Source, on the other hand, has as its main goal to optimise the reuse of critical assets within the organisation. After recognising a suitable software component as a critical asset, an organisation may strategically decide to manage it as an Inner Source project.

3.3.3.2 Support

As a result of the reuse focus, there is typically a difference in the level of support that is provided. In the infrastructure-based model, where projects are shared as the project’s initiator/creator sees fit, the level of support is dependent on the community that the project attracts. Support is therefore optional and very dependent on the project. Some projects may attract a lot of interest, whereas others may not. The project-based model, on the other hand, requires that there is support for the shared asset, since it is
part of the organisation’s strategy of planned reuse. Without support, business units, development teams or projects that use the shared asset may run into too many difficulties. Without sufficient support, the business strategy may be jeopardised.

### 3.3.3.3 Typical interface

Distribution and deployment of the software in the infrastructure-based model is typically done through a forge or portal, similar to Sourceforge.net (the largest online repository hosting OSS projects). A forge is a collaboration platform to allow collaborative software development over a network\(^{14}\). The portal provides the interface to the potential users, in which projects may be found. A forge platform aggregates a set of applications with integrated Web interfaces, and generally hosts multiple independent projects (Wikipedia, 2011). Project-based Inner Source typically has a well-known web page on the organisation’s intranet. Distribution of the shared asset’s source code can be straightforward through the use of a version control system, such as Subversion (SVN) or Git.

### 3.3.3.4 Owner/maintainer

Software projects in the infrastructure-based model are “owned” and maintained by the individual project’s creators/owners\(^{15}\). Maintenance is therefore dependent on the maintainer of the project and the community that the project attracts (similar to support), who may be busy with his normal development activities. In the project-based approach, there is typically a separate core team, which has been established in the organisation as an independently funded group, that has formal ownership of the shared asset.

### 3.3.3.5 Contributors

Contributions in the infrastructure-based model can come from anyone in the organisation who is interested in the project. While this can also be

---

\(^{14}\) http://en.wikipedia.org/wiki/Forge_(software)

\(^{15}\) That is to say, project owners decide where to take the project. The source code is usually formally owned by the organisation if it is written during work hours.
true for the project-based model, contributions would typically come from the core team, which has formal ownership and development and support responsibility for the shared asset, as well as customer project teams that integrate the shared asset into their development product.

3.3.3.6 Type of software

The types of software that is made available in the infrastructure-based model are typically packages such as tools and utilities, including compilers and shells, or basic functionality such as an XML parser. Such tools or other utilities are shared throughout the company in order to allow others reuse it and save efforts of having to recreate such tools. In the project-based model, the type of software typically comprises business-critical assets that are essential to the products being developed, which require more domain knowledge to produce. Such an asset could be for instance the platform in a software product line (SPL).

3.4 THE ADOPTION OF OSSD PRACTICES IN INDUSTRY

There have been a number of reports of organisations that have adopted Inner Source. Table 3-3 provides an overview of the organisations (as well as references to the reports), the term used at each organisation, the scope of openness (e.g., organisation, or a consortium of partners), and the organisations’ motivations to adopt Inner Source, as reported in the specified references.

Not all initiatives can be classified as infrastructure- or project-based. The third tier of HP’s POS model is Open Source, which is why the classification does not apply. Too little information is available for DTE Energy and Kitware to be able to classify them as either infrastructure or project-based.
Table 3-3. Overview of reported cases of Inner Source adoption.

<table>
<thead>
<tr>
<th>Organisation</th>
<th>Term (reference)</th>
<th>Model</th>
<th>Scope</th>
<th>Motivation (as reported in references)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hewlett-Packard</td>
<td>Progressive Open Source (Dinkelacker et al., 2002)</td>
<td>Inner Source, Corporate Source initiative</td>
<td>Infrastructure-based</td>
<td>Organisation</td>
</tr>
<tr>
<td></td>
<td>Controlled Source, Collaborative Development Program</td>
<td>Infrastructure-based</td>
<td>Consortium/selected business partners</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Open Source</td>
<td>N/A</td>
<td>Anyone with an Internet connection</td>
<td></td>
</tr>
<tr>
<td>Alcatel-Lucent</td>
<td>Corporate Open Source (Gurbani et al., 2006; Gurbani et al., 2010)</td>
<td>Project-based</td>
<td>Organisation</td>
<td>In order to speed development and quickly add functionality desired by different project groups who wanted to make use of it in their product lines.</td>
</tr>
<tr>
<td>Philips Healthcare</td>
<td>Inner Source, Inner Source Software (Wesselius, 2008)</td>
<td>Project-based</td>
<td>Organisation or consortium</td>
<td>To address issues such as catching up with existing software components and aligning the platform road map with product groups’ needs.</td>
</tr>
<tr>
<td>Nokia</td>
<td>Inner Source, iSource initiative (Lindman et al., 2008)</td>
<td>Infrastructure-based</td>
<td>Organisation or consortium</td>
<td>The idea behind iSource was to provide a portal enabling visibility of software and the source code. The goals were to increase individual engineers’ awareness of software developed inside the company, and to boost innovation by avoiding the problem of re-implementing the wheel.</td>
</tr>
<tr>
<td>Organisation</td>
<td>Initiative/Method</td>
<td>Collaboration Model</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------</td>
<td>---------------------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>SAP</td>
<td>SAP Forge initiative (Riehle et al., 2009)</td>
<td>Infrastructure-based</td>
<td>To make research-to-product successes like Gurbani’s [at Alcatel-Lucent] server software happen more often and smoothly.</td>
<td></td>
</tr>
<tr>
<td>IBM</td>
<td>Community Source (Taft, 2009; Taft, 2005)</td>
<td>Infrastructure-based</td>
<td>IBM officials said community source provides an infrastructure for nurturing componentization and facilitating reuse of those components.</td>
<td></td>
</tr>
<tr>
<td>DTE Energy</td>
<td>Open Source Methods (Alter, 2006)</td>
<td>N/A</td>
<td>One of our objectives is to build a software component library that accelerates code reuse. That reduces cost and increases the speed at which we can deliver new capabilities to the business.</td>
<td></td>
</tr>
<tr>
<td>Microsoft</td>
<td>OfficeLabs (Asay, 2007); CodeBox (Ogasawara, 2008)</td>
<td>Infrastructure-based</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>Kitware</td>
<td>Not specified. (Martin and Hoffman, 2007)</td>
<td>N/A</td>
<td>[The] development approach we use must have low enough overhead that we can use it with smaller projects but also be able to handle larger projects consisting of thousands of source files. [...] plus the need to cross-develop on Windows, Linux and Mac OSX. [...] With the variety of open source and closed-source tools available, a small organization can build high-quality robust software and save time in the process.</td>
<td></td>
</tr>
</tbody>
</table>
3.5 NORMATIVE GUIDELINES TO ADOPT INNER SOURCE

A few authors have reported guidelines and frameworks for the adoption of Inner Source. These are normative guidelines, as they prescribe what ought to be done, or preconditions that ought to be complied with. This section provides an overview.

3.5.1 Raymond’s preconditions for bazaar style development

In his seminal book “The Cathedral and the Bazaar”, Raymond (2001) provides an extensive analysis of how OSS development works in practice. Based on his analysis and extensive experience as an OSS developer, he lists a number of preconditions for a bazaar style development environment to work successfully. Note that he considers bazaar style to refer to OSS style development, and that his advice is not necessarily applicable to a commercial software-development organisation as-is. These preconditions may have to be tailored to fit a commercial environment (i.e., Inner Source). Raymond’s preconditions are listed in Table 3-4.

Table 3-4. Raymond’s preconditions for bazaar-style development.

<table>
<thead>
<tr>
<th>ID</th>
<th>Precondition</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>Your nascent developer community needs to have something runnable and testable to play with.</td>
</tr>
<tr>
<td>R2</td>
<td>You need to present a plausible promise. Your program should not fail to: a) run, and b) convince potential co-developers that it can be evolved in something really neat in the foreseeable future.</td>
</tr>
<tr>
<td>R3</td>
<td>It is absolutely critical that the coordinator be able to recognize good design ideas from others.</td>
</tr>
<tr>
<td>R4</td>
<td>Don’t make too clever designs; rather, keep them robust and simple in order to keep the software stable.</td>
</tr>
<tr>
<td>R5</td>
<td>A bazaar coordinator or leader must have good people and communication skills</td>
</tr>
</tbody>
</table>

3.5.2 Lessons learnt by Neus and Scherf

Neus and Scherf (2005) emphasised that introducing open source collaboration methods require a cultural change. They conducted workshops at different organisations to inform people of open,
collaborative ways of working, and to show the benefits. Based on their experiences they present five lessons learnt, summarised in Table 3-5.

Table 3-5. Lessons learnt to introduce open source collaboration methods, according to Neus and Scherf (2005).

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N1</td>
<td>Keep it simple. Do not overload people with methodology or jargon.</td>
</tr>
<tr>
<td>N2</td>
<td>Find passionate people. To drive change, you need passion. You need people who understand and are excited about the change.</td>
</tr>
<tr>
<td>N3</td>
<td>Do the “emperor’s clothes” test on the organisation to see how open or hierarchic the organisation is.</td>
</tr>
<tr>
<td>N4</td>
<td>Involve me, and I will understand. Cultural change cannot be forced; it can only be facilitated. Nothing is as powerful a teacher as first-hand experience. Allow people to experience what open-source collaboration could mean for them in their working environment.</td>
</tr>
<tr>
<td>N5</td>
<td>Start small, grow fast. Start small with a limited scope and the mission to solve a concrete problem. Demonstrate value, then grow.</td>
</tr>
</tbody>
</table>

3.5.3 Sharma et al.’s framework

Sharma et al. have explored “how organisations can foster an environment similar to OSS to manage their software development efforts” (Sharma et al., 2002). They have analysed OSS communities using an established theoretical framework based on three dimensions in the organisational theory literature: structure, process and culture. The characteristics are contrasted with “traditional organisations”, and transition steps are proposed to create a “hybrid OSS” community. The transition steps proposed by Sharma et al. are listed in Table 3-6.
Table 3-6. Transition steps to hybrid-OSS according to Sharma et al. (2002)

<table>
<thead>
<tr>
<th>Dim.</th>
<th>Element</th>
<th>ID</th>
<th>Proposed transition step</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Organisational structure</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Division of labour</td>
<td>S1</td>
<td>Voluntary, skill, competence and knowledge-based task assignment.</td>
</tr>
<tr>
<td></td>
<td>Coordination mechanisms</td>
<td>S2</td>
<td>Peer-based monitoring and reputation-based membership management and rules and institutions.</td>
</tr>
<tr>
<td></td>
<td>Decision rights</td>
<td>S3</td>
<td>Consensus-based, decentralized decision-making.</td>
</tr>
<tr>
<td></td>
<td>Boundaries</td>
<td>S4</td>
<td>Permeable, governed by organisational rules and institutions.</td>
</tr>
<tr>
<td></td>
<td>Importance of informal structure</td>
<td>S5</td>
<td>Informal networks</td>
</tr>
<tr>
<td></td>
<td>Politics</td>
<td>S6</td>
<td>Issues of concern to organisational members.</td>
</tr>
<tr>
<td></td>
<td>Basis of authority</td>
<td>S7</td>
<td>Reputation, based over a sufficiently long period</td>
</tr>
<tr>
<td></td>
<td>Governance</td>
<td>S8</td>
<td>Empowerment</td>
</tr>
<tr>
<td></td>
<td>Membership management</td>
<td>S9</td>
<td>Select qualified people, assign tasks</td>
</tr>
<tr>
<td></td>
<td>Rules and institutions</td>
<td>S10</td>
<td>Facilitate creation of rules and norms, provide autonomy</td>
</tr>
<tr>
<td></td>
<td>Monitoring and sanctions</td>
<td>S11</td>
<td>Support open, peer monitoring</td>
</tr>
<tr>
<td></td>
<td>Reputation</td>
<td>S12</td>
<td>Provide authority to enforce sanctions and rewards</td>
</tr>
<tr>
<td></td>
<td>Development</td>
<td>S13</td>
<td>Recognize quality of work and promote reputation</td>
</tr>
<tr>
<td></td>
<td>Communication</td>
<td>S14</td>
<td>Idea generation; opportunity identification</td>
</tr>
<tr>
<td></td>
<td>Reward</td>
<td>S15</td>
<td>Assign tasks to people with appropriate background</td>
</tr>
<tr>
<td></td>
<td>Motivation</td>
<td>S16</td>
<td>Move away from current development methodology</td>
</tr>
<tr>
<td></td>
<td>Information</td>
<td>S17</td>
<td>Support independent code development and testing</td>
</tr>
<tr>
<td></td>
<td>Decision making</td>
<td>S18</td>
<td>Provide repository management and project coordination infrastructure</td>
</tr>
<tr>
<td></td>
<td>Trust</td>
<td>S19</td>
<td>Allocate resources for product release and support</td>
</tr>
<tr>
<td></td>
<td>Communication</td>
<td>S20</td>
<td>Encourage electronic communication among groups across locations</td>
</tr>
<tr>
<td></td>
<td>Reward</td>
<td>S21</td>
<td>Recognition of expertise</td>
</tr>
<tr>
<td></td>
<td>Motivation</td>
<td>S22</td>
<td>Broaden set of motivations</td>
</tr>
<tr>
<td></td>
<td>Information</td>
<td>S23</td>
<td>Information sharing across groups/teams openly</td>
</tr>
<tr>
<td></td>
<td>Decision making</td>
<td>S24</td>
<td>Self-management of autonomous knowledge workers</td>
</tr>
<tr>
<td></td>
<td>Trust</td>
<td>S25</td>
<td>Core developers need to work closely with one another and develop trust</td>
</tr>
</tbody>
</table>

Based on analysis and proposed transition steps, Sharma et al. have proposed a second framework for creating hybrid-OSS communities. This framework is shown in Table 3-7.
Table 3-7. Framework for creating a hybrid-OSS community, according to Sharma et al. (2002).

<table>
<thead>
<tr>
<th>Element</th>
<th>ID</th>
<th>Activity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Community building</td>
<td>F1</td>
<td>Build “community of practice”</td>
<td>Promotion of free exchange of ideas and information among their workers.</td>
</tr>
<tr>
<td></td>
<td>F2</td>
<td>Free flow of information</td>
<td>Allow workers to leverage the knowledge of others and identify potential opportunities for new innovations.</td>
</tr>
<tr>
<td></td>
<td>F3</td>
<td>Informal network</td>
<td>Get rid of high degree of formal structure and provide mechanisms for workers to complete tasks through informal relationships and networking.</td>
</tr>
<tr>
<td></td>
<td>F4</td>
<td>Electronic communication</td>
<td>Encourage electronic communication among groups across locations.</td>
</tr>
<tr>
<td>Community governance</td>
<td>F5</td>
<td>Shared-governance</td>
<td>1. Manage community of practice in a way that is perceived as fair and equitable, in order to sustain motivation.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2. Move away from imposing central command and control structure on the community.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3. Allow community members to work in teams and empower them to make decisions by discussion and voting.</td>
</tr>
<tr>
<td></td>
<td>F6</td>
<td>Shared membership management</td>
<td>4. Provide mechanisms for qualified people to join the community and contribute to the project.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>5. Allow new members to join and current members to leave.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>6. Allow members to forge and dissolve relationships with outside entities (e.g., customers, suppliers, vendors) as they see fit.</td>
</tr>
<tr>
<td></td>
<td>F7</td>
<td>Shared incentives and rewards</td>
<td>Develop a performance and measurement system, which rewards and promotes members based on meeting both community and organisational goals.</td>
</tr>
<tr>
<td>Community infrastructure</td>
<td>F8</td>
<td>Repository management</td>
<td>Provide infrastructure to store and manage artefacts.</td>
</tr>
<tr>
<td></td>
<td>F9</td>
<td>Version control</td>
<td>Before committing changes to artefacts (e.g., source code) they need to be evaluated by peers for quality and generality.</td>
</tr>
<tr>
<td></td>
<td>F10</td>
<td>Release coordination</td>
<td>Mechanisms for product release and documentation must be in place.</td>
</tr>
<tr>
<td></td>
<td>F11</td>
<td>Development and testing tools</td>
<td>Provide necessary tools and infrastructure for software development and project management.</td>
</tr>
</tbody>
</table>

Sharma et al. proposed the framework shown in for adopting hybrid-OSS communities based on three major elements: (1) community building, (2) community governance, and (3) community infrastructure.
3.5.4 Adopting OSSD practices through adopting OSSD tools

Robbins (2005) suggests that adoption of OSS development tools may influence software development processes that are used by software-developing organisations. One such organisation that has adopted certain OSS development practices through the adoption of OSS tools is Kitware (Martin and Hoffman, 2007).

Robbins discusses common OSS development practices and contrasts them with traditional practices. Table 3-8 lists these practices discussed by Robbins, using the same classification into categories as presented in (Robbins, 2005).

After establishing these tensions with traditional development approaches, Robbins continues by discussing a number of common OSS development tools. Some tools are: version control systems (e.g., Subversion), issue trackers (e.g., Trac), mailing lists, wikis, build tools (e.g., Make), design tools (e.g., ArgoUML), quality assurance tools (e.g., Lint, a static source code analyser), and Collaborative Development Environments (CDEs, e.g., SourceForge). An overview of some OSS tools is also presented in (Koranne, 2010). Robbins notes that, though there is a wide range of OSS tools available, for some traditional development activities no OSS tools seem to be available. For instance, requirement management, project management and scheduling are activities for which there are no OSS tools available. Robbins explains that this lack of tools is understandable, since these activities are not common in OSS projects. One could state that no OSS developer ever had this “itch” to scratch.
Table 3-8. OSS development practices and their tensions with traditional approaches, according to Robbins (2005).

<table>
<thead>
<tr>
<th>Category</th>
<th>ID</th>
<th>OSS Practice</th>
<th>Tension with traditional approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tools &amp; Community</td>
<td>O1</td>
<td>Provide universal, immediate access to all project artefacts, allowing anyone can participate; information widely available and is up-to-date.</td>
<td>Common to find printed binders of requirements that become out of date rapidly; use of tools that do not scale well to multisite projects, limited number of tool licenses. Some communication happens face-to-face, which is not shared throughout the company.</td>
</tr>
<tr>
<td></td>
<td>O2</td>
<td>Staff projects with motivated volunteers. Motivation comes in many forms in OSS. Developers typically self-select their tasks.</td>
<td>Traditional project developers have additional motivations: professionalism and financial incentives. Resources are limited and it is up to management to determine how those resources are allocated.</td>
</tr>
<tr>
<td></td>
<td>O3</td>
<td>Work in communities that accumulate software assets and standardize practices.</td>
<td>It is common for traditional projects to acquire licenses for specific tools using project-specific budget, with little standardization across projects.</td>
</tr>
<tr>
<td>Open Systems Design</td>
<td>O4</td>
<td>Follow standards to validate the project, scope decision making and enable reuse</td>
<td>The marketplace often demands that new products differentiate themselves from existing offerings by going beyond current standards.</td>
</tr>
<tr>
<td></td>
<td>O5</td>
<td>Practice reuse and reusability to manage project scope.</td>
<td>Traditional development teams are optimizing returns on their current project; the cost of providing on-going support for reusable components can be at odds with that goal.</td>
</tr>
<tr>
<td></td>
<td>O6</td>
<td>Support diversity of usage and encourage plurality of authorship.</td>
<td>Traditional development tools tend to have more platform restrictions making it harder for large organisations to select uniform tools. Product features are managed in an effort to maximize returns and minimizing support costs. Tasks are assigned to people who are held accountable for them, limiting their contributions. Cost of support for any contributed code may prevent its integration.</td>
</tr>
<tr>
<td>Planning &amp; Execution</td>
<td>O7</td>
<td>Release early, release often.</td>
<td>The traditional waterfall model tightly coordinates work and minimizes number of releases. Iterative development methods must still achieve enough functionality for a 1.0 release.</td>
</tr>
<tr>
<td></td>
<td>O8</td>
<td>Place peer review in the critical path. Peer-review has been shown to be one of the most effective ways to eliminate defects in the code.</td>
<td>Unlikely to be adopted in critical path unless project is developing a safety-critical system. Traditional peer-reviews require time for individual study of the code followed by a face-to-face review meeting.</td>
</tr>
</tbody>
</table>
Robbins concludes by analysing how the use of OSS tools could affect the software development practices in an organisation. This discussion is summarised in Table 3-9.

Table 3-9. Impact of adopting OSS development tools, according to Robbins (2005).

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>Free tools. Tools are free and support casual use; more developers will be able to access and contribute to artefacts in all phases of development. This may lead to better technical understanding, increasing productivity and improved quality.</td>
</tr>
<tr>
<td>B2</td>
<td>Open source. The source to all artefacts is available and up-to-date, resulting in less wasted effort due to out-dated information, and less rework on downstream artefacts.</td>
</tr>
<tr>
<td>B3</td>
<td>Open development. Casual contributors are supported in the development process, allowing non-developer stakeholders (e.g., management, sales, marketing and support) to participate more constructively in the project. This can help refine requirements and better align expectation, increasing customer satisfaction.</td>
</tr>
<tr>
<td>B4</td>
<td>Incremental development. Many of the tools support incremental releases, allowing teams to produce releases more regularly. Early releases help manage project risk and set expectations. It also adds the benefit of allowing rapid reaction to changing market demands.</td>
</tr>
<tr>
<td>B5</td>
<td>Task automation. Many of the tools aim to reduce unlikable work, which potentially increases productivity, as well as a faster time-to-market and increased customer satisfaction.</td>
</tr>
<tr>
<td>B6</td>
<td>Peer-review. Many tools address peer-review, allowing projects to catch more defects in review, or conduct reviews more frequently.</td>
</tr>
<tr>
<td>B7</td>
<td>Increased reuse. Project web sites and accessible artefacts provide access to the status and technical details. Other projects may more readily evaluate and select these components for reuse. HOWTOs, FAQs, mailing lists and issue trackers help to cost-effectively support reused components. Expected benefits include faster time-to-market, lower maintenance costs and improved quality.</td>
</tr>
<tr>
<td>B8</td>
<td>Community building benefits. Collaborative Development Environments (CDE), such as SourceForge, help to establish communities, and offer both short- and long-term benefits. In the short term, administrative and training costs of using powerful tools are reduced and they allow practical and secure access to the development artefacts. Long-term benefits include accumulation of development knowledge in a durable and accessible form, increased quality and reuse, and more consistent adoption of the organisation’s chosen methodology.</td>
</tr>
</tbody>
</table>

3.5.5 Torkar et al.’s opportunities to adopt OSSD practices

Torkar et al. (2011) compared OSS and industrial development methods, and distilled six opportunities for organisations to adopt OSSD practices. These are listed in Table 3-10; not all of them are common OSS practices, which is why they are discussed in brief next.
Table 3-10. Guidelines by Torkar et al. (2011).

<table>
<thead>
<tr>
<th>ID</th>
<th>OSS development practice</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>Reduce technical debt</td>
</tr>
<tr>
<td>T2</td>
<td>Define an entry path for newcomers</td>
</tr>
<tr>
<td>T3</td>
<td>Increase information availability and visibility</td>
</tr>
<tr>
<td>T4</td>
<td>Embrace asynchronous tools for communication and decision-making</td>
</tr>
<tr>
<td>T5</td>
<td>Let practitioners influence ways of working</td>
</tr>
<tr>
<td>T6</td>
<td>Allow task selection</td>
</tr>
</tbody>
</table>

3.5.5.1 Reduce technical debt

Technical debt is built up, for instance, when deadlines demand quick solutions, rather than elegant designs. As technical debt accumulates, the architecture deteriorates and it becomes harder to maintain and debug the code. There are two strategies to deal with technical debt.

1. First, situations that create technical debt should be kept at a minimum. In OSS projects, the code is subject to public exposure, which is a cheap and effective way to invoke some benefits of code reviews.
2. The second strategy is to plan in advance for technical debt. While activities to reduce the technical debt are small compared to the main development efforts, they need to be isolated in order to separate them from development activities before release deadlines.

3.5.5.2 Define an entry path for newcomers

It is important to have a predefined path to allow new developers to learn while doing productive activities. If this is unattended, there is a risk that new developers are placed in positions for which they are unqualified, or making their learning curve unnecessarily long. Suggested activities include bug fixing and reducing technical debt.

3.5.5.3 Increase information availability and visibility

Knowledge sharing between departments is important for developers to become aware of work performed elsewhere in the organisation (and to prevent redundant work). In order to improve information availability and visibility, it is important that information is easy to locate. A tool that indexes information sources and allows for centralised searching can support this. Read-access of information from other departments allows developers to increase reuse, not only of software components but also technical know-how. Public exposure would also favour modularised
designs, allow for external contributions and increase overall code quality. In order to protect intellectual property (IP), the authors propose a blacklist approach to lock certain IP sensitive assets. Finally, collaboration on creation and editing certain documents by making these documents available on the network may gather interested engineers and may improve the quality of such artefacts.

3.5.5.4 *Embrace asynchronous tools for communication and decision-making*

OSS projects heavily use asynchronous communication technologies (e.g., mailing lists). Such technologies allow individuals to go through a larger amount of information without blocking their attention at any time. There is no need to schedule meetings, which are time-consuming and difficult to scale up. With communication technologies, a larger number of interested parties can participate, allowing them to give feedback in the decision-making process. Furthermore, electronic communication allows for automatic archiving without loss of information. This is of particular importance for design and implementation decisions, which may have to be referenced and retrieved later. This archive also forms a useful knowledge base that can be used to lower the learning curve for newcomers. The knowledge is stored permanently, and independent of leaving of key employees.

3.5.5.5 *Let developers influence ways of working*

In OSS development, developers have a great deal of autonomy in choosing the process, methods and tools. In traditional software development, this may have been predefined. Developers are the people who are doing the development work. As such, they should have a say in how development takes place, by openly discussing and suggesting improved ways of working.

3.5.5.6 *Allow task selection*

Providing a way to developers to channel their curiosity and self-development may help to improve motivation. This can be done by giving developers opportunities to let them choose work that they like to do. One
approach is to let employees dedicate part of their weekly working hours to a pool of diverse available tasks that cross the project and department boundaries. A few considerations must be made:

- It is important that there are always active engineers in a project at all times because contributors will need support.
- There needs to be a permanent knowledge base that outside contributors can consult without disturbing engineers with FAQs.

The task pool must contain activities of a size that are feasible to conduct during an employee’s ‘free time’ (part of his/her weekly working hours). It should also prevent rework by reflecting when someone is already working on a task. Developers should be able to propose new tasks. A resident mentor who helps new contributors might lower the learning curve and ensure that contributions fit in.

3.5.6 Gurbani et al.’s suggested software characteristics

Gurbani et al. (2006) suggest that certain commercial projects can benefit from open source development methodology, and identified a number of features. OSSD methodology was found to be especially suitable for projects that exhibit these features, which are listed in Table 3-11.

Table 3-11. Features of suitable projects for an open source development style, according to Gurbani et al. (2006).

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>G1</td>
<td>The technology is needed by several product groups (a common asset, and therefore a need to pool resources).</td>
</tr>
<tr>
<td>G2</td>
<td>The technology is relatively immature so that requirements and features are not fully known at the outset (and therefore there is a need to evolve continuously).</td>
</tr>
<tr>
<td>G3</td>
<td>Product groups have different needs and specific expertise in customising the software for their needs (so everyone benefits from the contributions of each group).</td>
</tr>
<tr>
<td>G4</td>
<td>The initial product has a sound modular architecture (which makes it feasible to merge all the diverse changes into a single development branch).</td>
</tr>
</tbody>
</table>

3.6 OVERVIEW OF GUIDELINES

The previous sections presented the various guidelines and suggestions for adopting OSS development practices by a number of authors. Each of these sets of guidelines address one or more aspects that the authors have
considered to be important, such as characteristics of a product, practices and processes that should be considered, the adoption of appropriate tools, and so on. Some authors have focused on one particular aspect; for instance, Robbins presents an analysis of the effect of the adoption of OSS tools on development practices, and Gurbani et al. list a number of “success factors” which are characteristics of the software product that is developed as a shared asset. Other researchers use their own classification, such as Sharma et al. They present an analysis of OSS and traditional software development using an established theoretical framework to analyse organisations based on three dimensions: structure, process and culture.

In seeking to integrate the studies reported above, the researcher has identified four high-level categories to which the studies above can be mapped (Table 3-12). Not all authors have explicitly identified such categories; for instance, Torkar et al. (2011) have identified six opportunities to adopt OSSD practices. Each of them can be classified in one of the categories in Table 3-12.

<table>
<thead>
<tr>
<th>Author</th>
<th>Product</th>
<th>Development practices</th>
<th>Tools &amp; Infrastructure</th>
<th>Organisation &amp; Community</th>
</tr>
</thead>
<tbody>
<tr>
<td>Raymond</td>
<td>Y</td>
<td></td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>Neus &amp; Scherf</td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sharma et al.</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Robbins</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>Torkar et al.</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gurbani et al.</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The table shows that different authors have considered different aspects; for instance, Raymond provides some “preconditions” for a bazaar-style way of working, which address aspects of the software that is managed as an “open source” product, but does not provide suggestions related to the development process or tools. Sharma et al. provide the most complete set of guidelines, but do not consider characteristics of the software product that is developed (the Inner Source Software).
It is important to note that the table provides a high-level overview only, and does not show how many guidelines each of the authors provide for each category. For instance, only one of the five “preconditions” by Raymond can be classified into the “People & Culture” category. Likewise, the table shows that both Sharma et al. and Torkar et al. provide guidance in three categories. However, the framework provided by Sharma et al. is much more extensive than the set of six guidelines provided by Torkar et al.

Though each of the reports presented by the various authors can be considered valuable to organisations that are interested in exploring the adoption of OSSD practices, a holistic set of guidelines that addresses all the important themes (product, process, etc.) is missing. For instance, Raymond’s preconditions consider the software product that would be managed as an OSS project (e.g., R1 in Table 3-4), but he does not consider development practices or tools to be used. Likewise, Neus and Scherf (see Table 3-5) only consider aspects of the organisation or community, but none regarding the product, development processes or tools. Sharma et al. do not provide any guidance related to the software product or the organisation and community. All considerations listed (see Table 3-7) refer to the development process and tools and infrastructure. All observations by Robbins (Table 3-8) consider development practices (e.g., O7, O8), tools (e.g., O1) and organisation & community (e.g., O2); none consider the software product. Torkar et al. (Table 3-10) also consider development practices (e.g., T1), tools & infrastructure (e.g., T4) and organisation & community (e.g., T6: the researcher considers task selection as work coordination, which is an organisational aspect), but not the software product. Finally, Gurbani et al. (see Table 3-11) only list “features” of a software product that would be suitable for OSS style development, but do not consider any of the other three categories.

3.7 CHAPTER SUMMARY

This chapter has presented an overview of the landscape of Inner Source research, which is still in its nascent phase. The chapter started with a
presentation of the two Inner Source models that have emerged (project-based and infrastructure-based). This was followed by an overview of Inner Source adoption that has been reported so far. The chapter also reviewed the literature that provides guidance for how to adopt OSS development practices. The review showed that, while various authors have proposed guidelines regarding a variety of aspects important in adoption of Inner Source, there has not been a holistic set of guidelines to advise organisations in this matter. To address this, the categorisation presented in Table 3-12 has been derived.
Chapter 4

The Role of Software Architecture in Using Software from the Bazaar

4.1 INTRODUCTION AND CHAPTER LAYOUT

Chapter 2 listed a number of scenarios in which OSS is used in industry, and highlighted two scenarios that are relevant to the research presented in this thesis, namely (1) the use of OSS products as components in Component-Based Software Development (CBSD), and (2) the use of OSS development practices. In both cases, software products are used in CBSD. In the first scenario, products are downloaded from the OSS projects’ websites, and integrated into a final product. In such a scenario, the software development organisation has the role of OSS integrator (Hauge et al., 2007). In the second scenario, software components are developed in an “internal” bazaar, namely, in an Inner Source environment (see Chapter 3), and subsequently integrated into end-products. In both scenarios, the component integrator (e.g., a product group that develops a product) is handling software developed by a community, which is either external to the organisation (Open Source) or internal (Inner Source).

Building systems from independently developed components is not without risks. Garlan et al. (1995) reported their experiences of using only four components and experienced significant integration problems. They identified the root cause of their problems to be architectural mismatch. That is, the architecture of the component does not match, or is not compatible with, the architecture of the system into which it is integrated. Software architecture is therefore recognised as a critical success factor in CBSD (Bosch and Stafford, 2002).
This chapter provides necessary background for the remainder of the thesis (in particular Chapters 8 and 9). It presents a brief overview of CBSD, software architecture (SA), the SA lifecycle, architectural knowledge, and the importance of SA within CBSD. Furthermore, this chapter reviews the literature on the role of software architecture in OSS; there is no literature on the role of software architecture in Inner Source.

The chapter starts with a brief introduction on CBSD (Section 4.2). The chapter continues with a brief overview of software architecture in general (Section 4.3), including a discussion of architectural patterns and architectural knowledge. This is followed by a brief review of the software architecture life cycle in Section 4.4. Since this life cycle does not explicitly consider the use of components, Section 4.5 discusses the software architecture life cycle in the context of CBSD. Section 4.6 provides an overview of the literature on software architecture in OSS research. Section 4.7 concludes the chapter.

### 4.2 COMPONENT-BASED SOFTWARE DEVELOPMENT

In the last 20 years or so, CBSD has become a common approach to build large-scale software systems (Szyperski, 2002; Wallnau et al., 2002). An organisation may use components from different sources: they may be acquired externally or developed in-house. In the former case, components may be Commercial Off-The-Shelf (COTS) or OSS products (Hissam and Weinstock, 2001), which have become viable alternatives to COTS components (Fitzgerald, 2006). The use of OSS components falls within the scope of this thesis.

When developing components in-house, components may be developed using “traditional” development methods, but as Chapters 2 and 3 have outlined, a number of organisations have started to adopt OSS development practices to develop components in-house. The latter option also falls within the scope of this thesis. Figure 4-1 presents a taxonomy.

---

16 The research acknowledges that there is no single traditional development method, which is why the word “traditional” is in quotation marks.
of Off-The-Shelf (OTS) components. The taxonomy in the figure uses UML class diagram notation to represent the different types of components (using “class inheritance”).

![Diagram showing classification of components based on where and how they are developed (in UML 2 notation).]

**Figure 4-1. Classification of components based on where and how they are developed (in UML 2 notation).**

Whereas the specialisation in the sub-tree on the left-hand side is about the question *where*, or *by whom* the component is developed (by a third-party vendor or an OSS community), the specialisation in the sub-tree on the right-hand side is about the question *how* the component is developed. That is, the component is developed following OSS development practices (Inner Source Software) or it is developed using “traditional” development methods.

As is indicated by the grouping in the figure, this thesis focuses on the integration of OSS components and ISS components, both of which are developed using similar development practices.

### 4.3 SOFTWARE ARCHITECTURE

Since the late 1980s, Software Architecture (SA) has emerged as an important sub-discipline in Software Engineering (SE) research (Shaw and Clements, 2006). Despite two decades of research in this field, however, there is no commonly accepted definition of “software
architecture”. One of the most-cited definitions is by Bass et al. (2003), which is also adopted in this thesis:

The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

SA as a product has been recognised as an important design artefact in software development activities, including analysis, design and evaluation activities, as well as implementation and evolution (Tang et al., 2010). Section 4.4 discusses these activities in more detail. A software system’s SA typically constitutes the design decisions about a system, such as the use of certain architectural styles or patterns, e.g., layers or model-view-controller (MVC).

4.3.1 Architectural patterns

Architectural patterns are common solutions to recurring design problems, and affect the system-wide quality attributes (QAs, sometimes referred to as a system’s “ilities”), such as performance and reliability (Bass et al., 2003). For instance, a “layered” architecture is likely to improve maintainability, as it facilitates a clear separation of concerns. However, passing large numbers of “messages” (e.g., function calls) up and down a layer “stack” may negatively impact performance. A variety of patterns have been documented in detail, for instance in the Pattern-Oriented Software Architecture (POSA) book series (e.g., (Buschmann et al., 1996)).

Architectural patterns are sometimes referred to as architectural styles. Avgeriou and Zdun (2005) point out that there are some philosophical differences between these two “schools of thought”, but in the context of this research, these are not relevant.

Architectural patterns are at a higher level of abstraction than design patterns, such as documented by the “Gang of Four” (GoF) in (Gamma et al., 1995). However, as Avgeriou and Zdun (2005) argue:
In general it is hard to draw the line between architectural patterns and design patterns. In fact, it depends heavily on the viewpoint of the designer or architect whether a specific pattern is categorised as an architectural pattern or a design pattern.

4.3.2 Architectural Knowledge

Information such as used patterns, design decisions, etc. is referred to as “Architectural Knowledge” (AK). It should be noted that there is no commonly accepted definition of AK within the SA community (de Boer and Farenhorst, 2008). In recent years, the SA research community has recognised the value of AK and has started to focus on recording, managing and sharing AK (Ali Babar et al., 2009a). An example of the efforts made by the SA community to study and evangelise the importance of AK is the SHARK (Sharing and Reusing Architectural Knowledge) workshop series, which has been organised since 2006 (Lago et al., 2010).

4.4 THE SOFTWARE ARCHITECTURE LIFE CYCLE

Software architecture is used in various architecting activities, spanning the initial stage of architecture creation to architecture evolution and maintenance (Tang et al., 2010). Hofmeister et al. (2007) identified a general model of software architecture design based on five industrial approaches to architectural design. This model consists of three activities: (1) architectural analysis, (2) architectural design, and (3) architectural evaluation. Tang et al. (2010) extended this model to include two other activities of the software architecture life cycle: (4) implementation and (5) maintenance. In their work, Tang et al. focus specifically on the role of Architectural Knowledge (AK) throughout the architectural life cycle. An adapted version of the extended architecture life cycle by Tang et al. is shown in Figure 4-2. Tang et al. did not include titles on the arrows between the life cycle’s stages as defined by Hofmeister et al. These titles are restored in the adapted version of the figure below.
The remainder of this section discusses each of the life cycle’s stages in more detail.

### 4.4.1 Architectural analysis

The architectural analysis stage is characterised by acquiring an understanding of the design problem that the architect needs to solve (Hofmeister et al., 2007). The architect should consider the architectural constraints as well as the context in which the system is to be built and used. This results in a set of architecturally significant requirements (ASRs), which provide input for the second stage, architectural synthesis, which is discussed next.

### 4.4.2 Architectural synthesis

In the architectural synthesis stage, the architect proposes one or more architecture solutions that satisfy the set of ASRs that resulted from the
architectural analysis stage. This is the core activity in the design phase, moving from the problem space into the solution space (Hofmeister et al., 2007).

Architecture designs must be documented using a certain notation in order to communicate and reason about the design. To that end, a number of Architecture Description Languages (ADLs) have been proposed. Medvidovic and Taylor (2000) have performed a framework-based comparison and classification of ten ADLs. A commonly used notation is the Unified Modelling Language\(^\text{17}\) (UML).

### 4.4.3 Architectural evaluation

After one or more architecture solutions have been generated, it is important that they are evaluated in order to make sure that the solution can satisfy the design problem and constraints sufficiently. To that end, a number of architectural evaluation methods have been proposed. The Software Engineering Institute (SEI) developed three methods: ATAM, SAAM and ARID, all of which are presented in (Clements et al., 2002). Dobrica and Niemelä (2002) performed a survey on software architecture analysis methods, and have done a framework-based comparison of eight methods. Ali Babar et al. (2004) also performed a framework-based comparison and classification of software architecture evaluation methods.

Despite this range of available architecture evaluation methods, a recent survey among practitioners showed that these methods are rarely used (Ali Babar and Gorton, 2009). Architectural evaluations are considered “expensive”, in that they require much time and involvement of stakeholders. This is troublesome in particular in smaller software-developing organisations that have few resources available to perform such evaluations. Harrison and Avgeriou (2011) have proposed the Pattern-Based Architecture Review (PBAR) method, which is a

\(^{17}\) Some authors consider the UML as an ADL; others disagree. This discussion is outside the scope of this thesis.
“lightweight” software architecture evaluation methods based on knowledge of a system’s architectural patterns.

4.4.4 Architectural implementation

After an architecture has been evaluated and found to be satisfactory to achieve the ASRs, the architecture is realised by designers and developers, possibly under supervision of the original architect. During the implementation phase, it is important to pay sufficient attention to the ASRs, which include requirements with respect to the system’s quality attributes, such as performance and reliability. Furthermore, it is also important that the implementation of the architecture conforms to the architectural design decisions that were made in the design phase.

4.4.5 Architectural maintenance

After the system has been deployed, it needs to be maintained. Maintenance is typically performed by others than the designers and developers who created the system in the first place. Therefore, it is important that sufficient architectural knowledge of the system is recorded and documented. If the AK of the system is no longer available, the cost of maintenance can increase significantly.

It is not uncommon, however, that a system’s AK is not available. This may be due to the fact that the AK was never recorded and stored in the first place; it had remained tacit knowledge. Researchers have long identified this problem, and various methods and tools have been proposed to recover such AK. Chikofsky and Cross (1990) presented a taxonomy of reverse engineering and design recovery. More recently, Tonella et al. (2007) presented the state-of-the-art of empirical studies in reverse engineering. Jansen et al. (2008) presented ADDRA, which is an approach to recover and document architectural design decisions.
4.5 ARCHITECTURE LIFE CYCLE FOR COMPONENT-BASED SOFTWARE DEVELOPMENT

So far, the discussion of the architectural life cycle (shown in Figure 4-2) has not explicitly acknowledged development with components, which are independently developed. This section discusses the role of software architecture in CBSD.

In CBSD, there is a distinction between the development life cycle for components and with components. The architecture life cycle for components is fundamentally the same as the “classic” life cycle, as shown in Figure 4-2; in this case, the component is the system that is being developed. The architecture life cycle for development with components is similar but different. Li et al. (2009) defined three main activities in CBSD: (1) component selection, (2) integration and (3) maintenance. It should be noted that this applies to CBSD with externally developed components; if components are developed in-house, they do not need to be selected.

Table 4-1 presents a mapping of the three phases of the “classic” life cycle to the development life cycle with components. These differences are discussed next. The left column lists the phases of the “classic” life cycle by Tang et al. (2010). The right column lists phases of the architectural life cycle for CBSD.

Table 4-1. Mapping of phases of the architecture life cycle.

<table>
<thead>
<tr>
<th>“Classic” architectural life cycle</th>
<th>Architectural life cycle for CBSD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design</td>
<td>Design and component selection</td>
</tr>
<tr>
<td>Implementation</td>
<td>Implementation and component integration</td>
</tr>
<tr>
<td>Maintenance</td>
<td>System and component maintenance</td>
</tr>
</tbody>
</table>

The researcher is aware that the mapping presented in Table 4-1 is somewhat “theoretical”; the “traditional” software development process is customised when using components (Li et al., 2006). In traditional processes, the design phase (as defined in Figure 4-2) includes requirements elicitation. Requirements elicitation can no longer be an isolated activity in CBSD, since integrators need to consider the
availability of suitable components, each of which may have or not have certain features (Wallnau et al., 2002). Therefore, selection of components becomes a trade-off process between a system’s requirements and the feature set of the candidate components.

4.5.1.1 Design and component selection

The classic architecture life cycle does not explicitly take into account the use of pre-fabricated components. Rather, it assumes a more classic software development approach, whereby the system is implemented from scratch. The design phase in the “classic” architecture life cycle (as defined in (Hofmeister et al., 2007)) contains three main activities, namely (1) architecture analysis, (2) architecture design and (3) architecture evaluation.

When building systems with components, the design phase is augmented with a selection phase, during which components are evaluated and selected. As in the classic life cycle, there is still an analysis phase, during which architectural significant requirements must be identified and analysed. However, besides synthesising architectural designs, in CBSD appropriate components should be identified, evaluated and selected that can fulfil the ASRs.

Selection of appropriate components is a critical success factor in CBSD, and both researchers and industry have proposed a variety of evaluation and selection methods for OSS\(^\text{18}\). However, studies have also shown that practitioners typically do not use these “normative” selection methods (Hauge et al., 2009; Merilinna and Matinlassi, 2006). Hauge et al. (2009) observed a first fit rather than a best fit strategy.

4.5.1.2 Implementation and component integration

The implementation of a system involves coding activities and component integration. Not all required functionality may be covered by components; some features may have to be implemented from scratch. Also, it is quite common to write so-called “glueware” (or “glue code”) to

\(^{18}\) Evaluation methods for COTS components have also been proposed, but since this thesis focuses on software “from the bazaar”, this is outside the scope of this thesis.
integrate components, or “addware” to complete the required functionality of components (Li et al., 2006). The source code’s availability of OSS (and ISS) components allow direct modification by the OSS integrator (Di Giacomo, 2005). However, integrators typically do not make changes to the source code, for various reasons. One obstacle is that it often requires an in-depth knowledge of the components to be able to make useful changes.

4.5.1.3 System and component maintenance
After system deployment, the system needs to be maintained. Maintenance in CBSD is also of a different nature than in non-CBSD, since each component evolves independently from the system in which it is integrated. That is, third-party component suppliers (including OSS communities) will continue to maintain a component (either through adaptive, corrective or perfective maintenance (Swanson, 1976)). A variety of issues may arise; for instance, new versions may not always be compatible with the rest of the system (Voas, 1998).

If custom changes were made during the integration phase, these may have to be separately maintained if the changes are not incorporated back into the source base that is maintained by the OSS community. An OSS integrator can choose to actively participate in the development or roadmap planning of the OSS product (Ven and Mannaert, 2008).

4.6 THE ROLE OF SOFTWARE ARCHITECTURE IN OPEN SOURCE SOFTWARE
Software architecture has received little attention in OSS research. This section provides a brief overview of research on SA in the context of OSS.

Tran et al. (2000) presented an experience report on “architectural drift”, when concrete architecture (implementation) differs from the conceptual architecture (design). They argue that developers can more easily understand a system if its architecture is repaired and demonstrate their approach to architectural repair for two OSS products. Nakagawa et al. (2008) presented a case study of an OSS web-based system, and also
found that the architecture had drifted after approximately two years of development. They highlight that the architecture affects the product quality, and propose architecture refactoring to repair the architecture.

Capiluppi and Knowles (2009) report on architecture recovery of three OSS products in the same domain (instant messaging) and found that a common architecture for these products had emerged. They argue that architecture recovery may facilitate OSS developers to understand the design as well as a sharing of tacit knowledge of other OSS developers.

Merilinna and Matinlassi (2006) investigated the state of the art and practice of OSS integration. While they did not specifically focus on SA, they found that architectural mismatch was either pre-empted by selection of OSS components from a list of “fluently integrating” components, or that practitioners were not aware of the concept of architectural mismatch.

Matinlassi (2007) performed a survey to investigate the role of SA as perceived by 15 OSS communities, and found that “architecture” was mostly considered to be a high-level, coarse grained abstraction of the system’s structure. Modularity was considered to be the most important QA of an SA, whereas other QAs such as performance and ease of integration were not considered important by most respondents.

Ali Babar et al. (2007) argued that organisations may be able to more confidently use OSS components in a software product line if OSS communities pay more attention to the architectural aspects during development and evolution of those components. Previously, researchers and practitioners have organised SA-related workshops (Ali Babar et al., 2009b) to draw attention to this topic in OSS, and proposed research roadmaps (Arief et al., 2001; Lennerholt et al., 2008). Recently, AK Management (AKM) in OSS projects is discussed from the perspective of OSS developers (Stamelos and Kakarontzas, 2010).

So far, there have only been a few empirical studies on the role and importance of SA in OSS. These mostly focused on architectural
repair and recovery. However, there has been no study that has focused on OSS integration from the OSS integrator’s point of view, and in particular with respect to a component’s AK.

4.7 CHAPTER SUMMARY

This chapter presented a brief overview of the concepts of component-based software development (CBSD), software architecture (SA), the SA lifecycle, architectural knowledge (AK), and the importance of SA within CBSD. This chapter provides the necessary background for the remainder of the thesis, in particular Chapters 8 and 9.

This chapter concludes the background section of this thesis. Chapter 5 which follows next revisits the research objectives stated in Chapter 1, after which six research questions are derived based on the background presented in Chapters 2 to 4.
5.1 INTRODUCTION AND CHAPTER LAYOUT

This chapter presents an overview of the research process for the research presented in this thesis. The research consisted of a number of studies, all of which were conducted in the context of two research objectives, which were stated in Chapter 1. The overall research objectives led to the generation of six research questions, each of which was addressed by one or more research studies. Some of the research studies defined more specific research questions in order to answer the more general research questions.

Punch (2006) offers a general framework for developing research proposals, and distinguishes between five levels of concepts and questions that vary in levels of abstraction. Together, they form an inductive-deductive hierarchy. This is shown in Figure 5-1.

![Figure 5-1. Punch’s inductive-deductive hierarchy (based on a visualisation by (Leshem and Trafford, 2007).](image-url)
The presentation of the research design follows Punch’s hierarchy. According to Punch, research areas are usually stated in one or a few words. While the term “Open Source Software” is certainly relevant, it does not suffice to also cover “Inner Source”. Instead, in this research the “Bazaar” metaphor introduced by Raymond (2001) is used. Therefore, the research area can be described as: “software developed in ‘bazaar-style’ development communities”.

The research topic falls within the research area, and is one aspect of the research area. A researcher can choose any topic within a research area. In this research, the research topic is: “the use of ‘bazaar-style’ developed software in product development”.

This chapter proceeds as follows. Firstly, Section 5.2 refines the research objectives, and defines and justifies the research questions based on a brief reiteration of the discussion in the previous three chapters.

This thesis addresses six research questions; due to this multi-faceted research focus, the researcher has used a variety of research methods. An overview of the research approaches, methods and data collection methods is presented in Section 5.3. Sections 5.4 to 5.9 discuss these approaches and methods in more detail for each of the six research questions. Section 5.10 concludes the chapter.

5.2 RESEARCH OBJECTIVES

This research was concerned with investigating and documenting the challenges related to product development with software components developed in bazaar-style communities. Raymond first introduced the metaphor “Bazaar” to refer to Open Source communities, as opposed to the more top-down planned commercial software development, which he compares to the way cathedrals were built in the Middle Ages. With the emergence of “bazaar” style development within organisations that was discussed in Chapter 3, the researcher argues that there are two flavours of “bazaars”, namely Open Source and Inner Source communities. Since both scenarios are comparable (as was argued in Chapter 2; see Figure
the researcher felt that both scenarios are worth investigating in more detail. This led to the following two research objectives:

1. To document challenges and to provide guidance in product development with OSS components.

2. To identify challenges in Inner Source and to provide insights into when and how Inner Source can thrive.

The research presented in this thesis can mostly be characterised as “unfolding” rather than “prestructured”, following Punch’s (Punch, 2006) continuum shown in Figure 5-2.

![Figure 5-2. Prestructured versus unfolding research (from (Punch, 2006)).](image.jpg)

Punch argues that the difference between prestructured and unfolding research is the timing of the structuring of the research. The research design evolved during the process of conducting the research. The figure shows that this type of research is characterised by general open-ended questions, a loose design, and data that are not prestructured. Furthermore, following Figure 5-2, most of the research reported in this thesis was of a qualitative nature. Only one study (a quasi-experiment) can be considered quantitative research.

The remainder of this section outlines how the main objective of this research led to the generation of a number of general research questions (see Figure 5-1).

As was mentioned above, Chapter 2 discussed the emergence of Open Source Software adoption, and discussed five adoption scenarios, one of which is the integration of OSS components in component-based software development (CBSD). While there have been a number of studies reporting on developing with OSS components, there has been no special focus on what challenges OSS integrators may encounter. This led
the researcher to define the first research question, which is addressed in Chapter 6:

\[ RQ1: \text{What are the challenges associated with integration of open source software products?} \]

A widely cited issue in CBSD with OSS is the selection of appropriate and suitable OSS components. While some guidance is provided in the literature by means of OSS evaluation methods, there is also evidence that suggests that practitioners do not use these methods, nor are they aware of them (Hauge et al., 2009). In order to investigate this issue further, research question two is defined as follows:

\[ RQ2: \text{What is the state of the art of OSS evaluation and selection methods?} \]

RQ2 is addressed in Chapter 7.

Chapter 4 highlighted the importance of software architecture in CBSD. However, so far, there has been little attention for architectural aspects in OSS component integration. It is informative to investigate the importance of architectural knowledge of OSS components; do OSS integrators really need such information? Hence, research question three is:

\[ RQ3: \text{What is the importance of architectural knowledge for OSS integrators?} \]

The findings from RQ3 (presented in Chapter 8) indicate that OSS integrators do, in certain cases, appreciate availability of architectural knowledge. In particular, information about the architectural structure, or patterns, was found to be quite interesting. Furthermore, the literature suggests that based on architectural patterns, important insights can be gained into a component’s quality attributes, such as performance and reliability. Harrison and Avgeriou proposed a Pattern Based Architecture Review (PBAR) method, and showed that this lightweight approach for evaluating an architecture was considered to be quite useful (Harrison and Avgeriou, 2011). However, there is no practical guidance on how to
identify architectural patterns in an OSS product. Therefore, RQ4 is defined as:

*RQ4: How can architectural patterns in OSS products be identified?*

Detailed design and results of RQ4 are presented in Chapter 9.

As was established in Chapter 2, Inner Source organisations not only have adopted OSS development practices, they are also integrators of components that were developed in a “bazaar-style” environment, similar to OSS components. Since little is known about this corporate bazaar style, it is informative to gain insights into the challenges that may occur in a corporate bazaar. Hence, research question 5 is:

*RQ5: What are challenges and approaches to mitigate them in Inner Source?*

Detailed design and results of RQ5 are presented in Chapter 10.

RQ5 aims to identify challenges that may be encountered in an Inner Source environment, as well as approaches to address those challenges. As such, these findings may help other organisations that have already adopted Inner Source. However, there is limited guidance on adopting Inner Source. A number of potential benefits of adopting Inner Source have been reported in the literature, which may motivate organisations to consider adopting Inner Source, but it is not quite clear how and when Inner Source may thrive. Though a number of authors have reported guidance to adopt a “bazaar” within an organisation’s boundaries (as presented in Chapter 3), each set of guidelines seems to focus on one or a few aspects, such as guidelines related to the software product, organisation or culture. There has been no holistic approach to assess if and when Inner Source can thrive in an organisation. Hence, the researcher defined research question as follows:

*RQ6: What are the key factors to consider in adopting Inner Source?*

Whereas the findings of RQ5 provide support for existing Inner Source organisations, RQ6 aims to provide support organisations that have an
interest in Inner Source, but have not yet adopted it. Detailed design of the study to address RQ6 and results are presented in Chapter 11.

An overview of the six research questions is shown in Figure 5-3. The research is organised in two strands, which correspond to the two types of bazaars, Open Source and Inner Source. Research questions 1 to 4 focus on Open Source. Research questions 5 and 6 focus on Inner Source.

Each research question is addressed following a certain research approach; the following sections present and justify the research approaches and methods.
5.3 OVERVIEW OF RESEARCH APPROACHES

A variety of research approaches were used to address the six research questions derived in the previous section. This section presents an overview and justification of the research approaches; both research methods and data collection methods are discussed.

It is important to select an appropriate research method to conduct research, since each research method has strengths and weaknesses. The choice of research method may depend on the *purpose* of the study (Marshall and Rossman, 1999), the *type of research question* asked (Shaw, 2002; Easterbrook *et al.*, 2008), or the *state of prior research* (Edmondson and McManus, 2007). These factors all affect the choice of research method, which is graphically shown in Figure 5-4.

![Figure 5-4. The choice of research method is affected by several factors, as argued by different authors.](image)

There is no commonly accepted taxonomy of research methods in the field of software engineering (SE) research. Several authors have proposed classifications of research methodology in SE, and a number of authors provide guidance on selecting a suitable research method. This section continues with providing an overview of research methods used in SE research (Sub-section 5.3.1), followed by an overview of the guidance provided in the literature (Sub-section 5.3.2), based on which the selected research methods used for this research are identified and justified (Sub-section 5.3.3).
5.3.1 Research methods in software engineering research

Glass et al. (2002) analysed the state of research in SE. They distinguish between Research Approach and Research Method. According to Glass et al., the Research Approach is the overall approach undertaken in performing the research (listed in Table 5-1), whereas the Research Method (listed in Table 5-2) is a more detailed level of research technique (Glass et al., 2002).

Table 5-1. Research approaches and their usage (from (Glass et al., 2002)).

<table>
<thead>
<tr>
<th>ID</th>
<th>Approach</th>
<th>Identified usage</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Descriptive</td>
<td>27.9%</td>
</tr>
<tr>
<td>DS</td>
<td>Descriptive system</td>
<td>8.1%</td>
</tr>
<tr>
<td>DO</td>
<td>Descriptive other</td>
<td>18.2%</td>
</tr>
<tr>
<td>DR</td>
<td>Review of the literature</td>
<td>1.6%</td>
</tr>
<tr>
<td></td>
<td>Evaluative</td>
<td>13.8%</td>
</tr>
<tr>
<td>ED</td>
<td>Evaluative-deductive</td>
<td>4.3%</td>
</tr>
<tr>
<td>EI</td>
<td>Evaluative-interpretive</td>
<td>&lt;1%</td>
</tr>
<tr>
<td>EC</td>
<td>Evaluative-critical</td>
<td>1.4%</td>
</tr>
<tr>
<td>EO</td>
<td>Evaluative-other</td>
<td>7.3%</td>
</tr>
<tr>
<td></td>
<td>Formulative</td>
<td>55.3%</td>
</tr>
<tr>
<td>FF</td>
<td>Formulative-framework</td>
<td>4.1%</td>
</tr>
<tr>
<td>FG</td>
<td>Formulative-guidelines/standards</td>
<td>4.3%</td>
</tr>
<tr>
<td>FM</td>
<td>Formulative-model</td>
<td>9.8%</td>
</tr>
<tr>
<td>FP</td>
<td>Formulative—process, method, algorithm</td>
<td>36.0%</td>
</tr>
<tr>
<td>FT</td>
<td>Formulative-classification/taxonomy</td>
<td>1.1%</td>
</tr>
<tr>
<td>FC</td>
<td>Formulative-concept</td>
<td>3.0%</td>
</tr>
</tbody>
</table>

It is noteworthy to mention that the categories and taxonomies used by Glass et al. are intended to cover not only the Software Engineering (SE) research field, but all three major computing fields, which also include Computer Science (CS) and Information Systems (IS) (Glass et al., 2002). As a result, as also noted by Glass et al., some listed research methods listed are hardly used (see Table 5-2), or unlikely to be used in SE research. Glass et al. examined papers from six journals:

- Information and Software Technology
- Journal of Systems and Software
- Software Practice and Experience
- IEEE Software
- ACM Transactions on Software Engineering and Methodology
- IEEE Transactions on Software Engineering
The study was performed over a five-year period from 1995 to 1999. Every fifth paper in each journal was examined, resulting in a classification of 369 papers in total.

Glass et al. (2002) based their classification of research approaches on work by other authors, and distinguish *descriptive*, *evaluative* and *formulative* research approaches. The descriptive category contains papers that describe a system (subcategory DS), or something other than a system (subcategory DO), for instance, an opinion piece. Papers that primarily presented a review of the literature were classified in the third subcategory (DR). The evaluative category has four subcategories: three for the major epistemologies identified by Orlikowski and Baroudi (1991), plus one subcategory for evaluative studies that do not use one of the other three approaches, such as opinion surveys. The “formulative” category contains six subcategories for a variety of entities that can be formulated (proposed) by researchers. Unfortunately, Glass et al. do not provide further details about what defines a study as “descriptive”, “evaluative”, or “formulative”.

Table 5-1 shows that 27.9% of the papers included in the analysis by Glass et al. were classified as *descriptive*, 13.8% was classified as *evaluative*, and the majority (55.3%) of papers was classified as *formulative*.

While most of the research methods in Table 5-2 can be classified as “primary research” methods, some can be classified as “secondary research” methods (e.g., LR). Secondary research is also known as *desk research*, and involves investigating and aggregating existing literature (Crouch and Housden, 2003, p.19). Since 2004, Systematic Review (SR) and Systematic Mapping Study (SMS) have emerged as popular methods for conducting secondary research in SE research after a number of researchers started advocating evidence-based software engineering (EBSE) (Kitchenham *et al.*, 2004). Primary research, on the other hand, typically gathers data through field studies (e.g., CS, SU in the table) or laboratory experiments (e.g., LS).
Table 5-2. Research methods and their usage (from (Glass et al., 2002)).

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
<th>Identified usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>AR</td>
<td>Action research</td>
<td>0%</td>
</tr>
<tr>
<td>CA</td>
<td>Conceptual analysis</td>
<td>43.5%</td>
</tr>
<tr>
<td>CAM</td>
<td>Conceptual analysis/mathematic</td>
<td>10.6%</td>
</tr>
<tr>
<td>CI</td>
<td>Concept implementation (proof of concept)</td>
<td>17.1%</td>
</tr>
<tr>
<td>CS</td>
<td>Case study</td>
<td>2.2%</td>
</tr>
<tr>
<td>DA</td>
<td>Data analysis</td>
<td>2.2%</td>
</tr>
<tr>
<td>DI</td>
<td>Discourse analysis</td>
<td>0%</td>
</tr>
<tr>
<td>ET</td>
<td>Ethnography</td>
<td>0%</td>
</tr>
<tr>
<td>FE</td>
<td>Field experiment</td>
<td>&lt; 1%</td>
</tr>
<tr>
<td>FS</td>
<td>Field study</td>
<td>&lt; 1%</td>
</tr>
<tr>
<td>GT</td>
<td>Grounded theory</td>
<td>&lt; 1%</td>
</tr>
<tr>
<td>HE</td>
<td>Hermeneutics</td>
<td>&lt; 1%</td>
</tr>
<tr>
<td>ID</td>
<td>Instrument development</td>
<td>0%</td>
</tr>
<tr>
<td>LH</td>
<td>Laboratory experiment (human subjects)</td>
<td>3.0%</td>
</tr>
<tr>
<td>LR</td>
<td>Literature review/analysis</td>
<td>1.1%</td>
</tr>
<tr>
<td>LS</td>
<td>Laboratory experiment (software)</td>
<td>&lt; 1%</td>
</tr>
<tr>
<td>MA</td>
<td>Meta-analysis</td>
<td>0%</td>
</tr>
<tr>
<td>MP</td>
<td>Mathematical proof</td>
<td>&lt; 1%</td>
</tr>
<tr>
<td>PA</td>
<td>Protocol analysis</td>
<td>0%</td>
</tr>
<tr>
<td>PH</td>
<td>Phenomenology</td>
<td>0%</td>
</tr>
<tr>
<td>SI</td>
<td>Simulation</td>
<td>1.1%</td>
</tr>
<tr>
<td>SU</td>
<td>Descriptive/exploratory survey</td>
<td>1.6%</td>
</tr>
</tbody>
</table>

As Table 5-2 demonstrates, there is a wide variety of research methods in use by SE researchers. Easterbrook et al. (2008) discuss a number of common research methods that they believe are most relevant in SE research\(^{19}\). The methods they consider are:

- Controlled Experiments (including Quasi-Experiments)
- Case Studies (both exploratory and confirmatory)
- Survey Research
- Ethnographies
- Action Research

Each research method has strengths and weaknesses, which suggests that one method is more appropriate in a certain situation than another. The research method indicates the overall strategy (e.g., a survey), within which the researcher may choose from a variety of data

\(^{19}\) It is interesting that the results of the analysis by Glass et al. do not seem to agree with this relevance. Controlled experiments (LH + LS) were only used in < 4% (3 + <1) of the papers; case study (CS) was only identified in 2.2% of the papers; survey research (SU) accounts for only 1.6% of the studies; no ethnographic studies (ET) were identified, nor was action research (AR) identified as a used method.
collection methods (e.g., interviews or questionnaires). Sometimes, authors do not clearly distinguish between research method and data collection method; Lethbridge et al. (2005) present an extensive discussion of data collection methods for “software field studies”, which could be interpreted as a more general term for any study performed outside the researcher’s office. Seaman (1999) presents an overview of methods for (amongst others) the collection of qualitative data, and does not mention any research methods within which such methods could be used (e.g., case study or survey). In the pilot systematic review that the researcher conducted prior to the doctoral research presented in this thesis (reported in (Stol and Ali Babar, 2009)), it was found that the majority of studies published in the International Conference on Open Source Systems did not distinguish between data collection and research method.

Besides data collection method, a researcher needs to consider the data analysis method; that is, the techniques used to analyse the collected data. Sometimes the data analysis approach is implied by the research method, as is the case for Grounded Theory (GT). Glaser and Strauss (1967) propose a number of data analysis methods as part of the GT approach, such as constant comparison. Data analysis methods are not presented in this chapter, however; these are presented as part of the research designs in each of the chapters that present the studies.

5.3.2 Guidance for selecting a research method

As was briefly mentioned above, several authors have argued that the choice of research method should be guided the purpose of the study (Marshall and Rossman, 1999), the type of research question (Shaw, 2002; Easterbrook et al., 2008), or the state of prior theory (Edmondson and McManus, 2007). Marshall and Rossman (1999) proposed a framework to match research purpose and strategy with research and data collection methods. Part of their framework is shown in Table 5-3.
Table 5-3. Classification of study purposes (according to Marshall and Rossman)

<table>
<thead>
<tr>
<th>Purpose of the study</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exploratory</td>
<td>To investigate little-understood phenomena. To identify or discover important categories of meaning. To generate hypotheses for further research.</td>
</tr>
<tr>
<td>Explanatory</td>
<td>To explain the patterns related to the phenomenon in question. To identify plausible relationships shaping the phenomenon.</td>
</tr>
<tr>
<td>Descriptive</td>
<td>To document and describe the phenomenon of interest.</td>
</tr>
<tr>
<td>Emancipatory</td>
<td>To create opportunities and the will to engage in social action.</td>
</tr>
</tbody>
</table>

Edmondson and McManus (2007) argue that the research design depends on the state of prior theory and research; they distinguish between three levels of maturity: nascent, intermediate, and mature. Based on the level of maturity, they classify the research design, including the type of research question and the type of data collected. Table 5-4 shows the type of research questions for each state of maturity.

Table 5-4. Levels of maturity and types of research questions (according to Edmondson and McManus).

<table>
<thead>
<tr>
<th>State of prior theory and research</th>
<th>Type of research questions</th>
<th>Type of data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nascent</td>
<td>Open-ended inquiry about a phenomenon of interest</td>
<td>Qualitative, initially open-ended data that need to be interpreted for meaning.</td>
</tr>
<tr>
<td>Intermediate</td>
<td>Proposed relationships between new and established constructs</td>
<td>Hybrid (both qualitative and quantitative).</td>
</tr>
<tr>
<td>Mature</td>
<td>Focused questions and/or hypotheses relating existing constructs</td>
<td>Quantitative data; focused measures where extent or amount is meaningful.</td>
</tr>
</tbody>
</table>

While Edmondson and McManus do not mention any research methods (e.g., case study, survey), they do provide suggestions regarding (amongst others) the type of data collected (i.e., qualitative, quantitative) and data collection methods (e.g., interviews, surveys\(^{20}\)).

Shaw (2002) analysed paper submissions to the 2002 edition of the International Conference on Software Engineering (ICSE), and identified a number of categories of research questions, research results,

---

\(^{20}\) The researcher notes that Edmondson and McManus consider “survey” to be a data collection method, rather than a research method.
It is important to point out that there is a potential bias in this classification, since it is based on submissions to one particular conference, as opposed to, for instance, the International Conference on Software Architecture (ICSA), which is a general conference that is not focused on one specific field within software engineering (as opposed to, for instance, the European Conference on Open Source Systems (OSS)).

Further guidance for selecting methods is provided by Easterbrook et al. (2008). Table 5-6 summarises the research stages and types of questions asked in each research stage, as discussed by Easterbrook et al. Their focus is mostly on the selection of methods for empirical (field) studies, the researcher argues that some types of questions asked in each research stage, as discussed by Easterbrook et al.
Table 5-6. Research stages and types of research questions (according to (Easterbrook et al., 2008)).

<table>
<thead>
<tr>
<th>Research stage</th>
<th>Type of question</th>
<th>Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exploratory</td>
<td>Existence questions</td>
<td>Does X exist?</td>
</tr>
<tr>
<td></td>
<td>Description and Classification</td>
<td>What is X like?</td>
</tr>
<tr>
<td></td>
<td>Classification questions</td>
<td>How does X differ from Y?</td>
</tr>
<tr>
<td>Base-rate</td>
<td>Frequency and distribution</td>
<td>How often does X occur?</td>
</tr>
<tr>
<td></td>
<td>questions</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Descriptive-Process questions</td>
<td>How does X normally work?</td>
</tr>
<tr>
<td>Relationship</td>
<td>Relationship questions</td>
<td>Are X and Y related?</td>
</tr>
<tr>
<td>Causality</td>
<td>Causality questions</td>
<td>Does X cause Y?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Does X prevent Y?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>What causes Y?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>What are all the factors that cause Y?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>What effect does X have on Y?</td>
</tr>
<tr>
<td></td>
<td>Causality-Comparative questions</td>
<td>Does X cause more Y than does Z?</td>
</tr>
<tr>
<td></td>
<td>Interaction</td>
<td>Does X or Z cause more Y under one condition but not others?</td>
</tr>
<tr>
<td>Knowledge</td>
<td>Design questions</td>
<td>What’s an effective way to achieve X?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>What strategies help to achieve X?</td>
</tr>
</tbody>
</table>

5.3.3 Overview of research methods in this thesis

The research presented in this thesis addresses six research questions, each of which addresses a different type of research question, has a different research purpose, and for which the existing theory is in a different state of maturity. Table 5-7 presents an overview of the research questions and research methods used to address them. The choice of research methods was based on the guidance provided in Table 5-1 to Table 5-6. For instance, the table shows that RQ1 is concerned with “OSS Challenges”; the purpose of the study (following the classification by (Marshall and Rossman, 1999)) was “descriptive”; the state of theory (following (Edmondson and McManus, 2007)) was “intermediate”; the type of research question following the classification by (Shaw, 2002)) is “generalisation or characterisation”, and “description and classification” according to Easterbrook et al. The researcher chose to conduct desk research to address this question. The research approach, using the classification scheme by Glass et al., is “Review of literature”. The researcher chose to use systematic mapping study as a research method.
Table 5-7. Overview of research questions and research methods.

<table>
<thead>
<tr>
<th>RQ</th>
<th>Topic</th>
<th>Purpose</th>
<th>State of theory</th>
<th>Type of research question (Shaw)</th>
<th>Research stage / Type of question (Easterbrook et al.)</th>
<th>Used research type</th>
<th>Used research approach (Glass et al.)</th>
<th>Used research method</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Identifying OSS Challenges</td>
<td>Descriptive</td>
<td>Intermediate</td>
<td>Generalisation or characterisation</td>
<td>Exploratory / Description and Classification</td>
<td>Desk research</td>
<td>DR (Literature review)</td>
<td>Systematic mapping study</td>
</tr>
<tr>
<td>2</td>
<td>Analysing OSS evaluation methods</td>
<td>Descriptive</td>
<td>Intermediate</td>
<td>Generalisation or characterisation</td>
<td>Exploratory / Descriptive-Comparative</td>
<td>Desk research</td>
<td>DR (Literature review)</td>
<td>Literature review + Comparative study</td>
</tr>
<tr>
<td>3</td>
<td>Architectural knowledge needs of OSS integrators</td>
<td>Exploratory</td>
<td>Nascent</td>
<td>Generalisation or characterisation</td>
<td>Exploratory / Description and Classification</td>
<td>Field research</td>
<td>EO (Evaluative Other)</td>
<td>Survey</td>
</tr>
<tr>
<td>4</td>
<td>Guidance for identifying Architectural Patterns</td>
<td>Exploratory</td>
<td>Nascent</td>
<td>Method or means of development</td>
<td>Knowledge / Design</td>
<td>Field research</td>
<td>FP (Formulative Process)</td>
<td>Improvement Paradigm</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Design, evaluation, or analysis of a particular instance</td>
<td>Causality / Causality</td>
<td>Field research</td>
<td>ED (Evaluative Deductive)</td>
<td>Quasi-experiment</td>
</tr>
<tr>
<td>5</td>
<td>Identifying challenges in Inner Source</td>
<td>Exploratory</td>
<td>Nascent</td>
<td>Generalisation or characterisation</td>
<td>Exploratory / Description and Classification</td>
<td>Field research</td>
<td>DO (Descriptive Other)</td>
<td>Case study</td>
</tr>
<tr>
<td>6</td>
<td>Providing guidelines to adopt Inner Source</td>
<td>Exploratory</td>
<td>Nascent</td>
<td>Method or means of development</td>
<td>Knowledge / Design</td>
<td>Desk + Field research</td>
<td>FF (Formulative Framework)</td>
<td>Literature review + Case study</td>
</tr>
</tbody>
</table>
Sections 5.4 to 5.9 presents the research approaches for each of the six research questions. For each research question, the choice of research and data collection method is justified. These sections do not, however, present detailed discussion of the research design for each research question; rather, these details are presented in each of the chapters that present the research question (Chapters 6 to 11).

5.4 RESEARCH APPROACH FOR RQ1: IDENTIFYING OSS CHALLENGES

The first research question is to investigate what are the challenges that integrators may face when building products with OSS components. To that end, rather than conducting field research, the researcher chose to conduct desk research (Crouch and Housden, 2003, p.19), since there was already a body of primary studies on the use of OSS components in product development.

There are different approaches to conduct literature reviews; most prominent are “traditional” (ad-hoc) literature reviews and the Systematic Literature Review (SLR), also referred to as a Systematic Review (SR). A traditional literature review is usually conducted following an unsystematic and ad-hoc approach; a researcher can start with a number of well-known or highly cited relevant papers, from which references are followed to identify other relevant studies. This approach is typically difficult to repeat, since the review is not guided by a protocol or systematic approach.

SR is one method to conduct secondary research, and has become a popular approach to review existing literature on a topic since its advocates introduced it as an important method to support Evidence-Based Software Engineering (EBSE) in 2004 (Kitchenham et al., 2004). Kitchenham and Charters (2007) describe a systematic literature review as follows:
A systematic literature review (often referred to as systematic review) is a means of identifying, evaluating and interpreting all available research relevant to a particular research question, or topic area, or phenomenon of interest.

A related type of systematic approach to review relevant literature is the Systematic Mapping Study (SMS), or mapping study for short. Kitchenham and Charters have proposed guidelines for conducting SRs in Software Engineering (Kitchenham and Charters, 2007). These guidelines are also suitable for conducting mapping studies, which are also known as scoping studies. While similar to systematic reviews, there are a few key differences with mapping studies (Kitchenham and Charters, 2007, p.44), which are summarised in Table 5-8.

Table 5-8. Key differences between systematic reviews and systematic mapping studies.

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Systematic reviews</th>
<th>Mapping studies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Research question</td>
<td>Narrow, focussed</td>
<td>Broad, wider focus</td>
</tr>
<tr>
<td>Search term</td>
<td>Highly focussed</td>
<td>Less focussed</td>
</tr>
<tr>
<td>Number of identified studies</td>
<td>Small number of studies</td>
<td>High number of studies</td>
</tr>
<tr>
<td>Data extraction process</td>
<td>Specific information needed to answer the research questions. Typically numerical data such as number of subjects, treatment effect, etc.</td>
<td>Broad, can be termed as “classification” or “categorisation”</td>
</tr>
<tr>
<td>Analysis</td>
<td>Typically in-depth analysis techniques.</td>
<td>Is about summarising the data to answer the research questions. Unlikely to include in-depth analysis.</td>
</tr>
<tr>
<td>Dissemination</td>
<td>Disseminated in journal or conference papers.</td>
<td>The results may be more limited than for a systematic review; limited to commissioning bodies and academic publications, with the aim of influencing the future direction of primary research.</td>
</tr>
</tbody>
</table>

Though the table above presents a characterisation of systematic reviews and mapping studies in order to highlight typical differences, it must be noted that in practice there is no clear-cut distinction between the two

21 The first version of these guidelines (published in 2004) was authored solely by Kitchenham.
types of reviews. Some studies that are presented as “systematic reviews” are, in fact, mapping studies. For instance, as mentioned in Chapter 1, prior to this doctoral research the researcher conducted a pilot systematic review, which was published in (Stol and Ali Babar, 2009; Stol et al., 2009). However, that study is (in hindsight) better characterised as a mapping study.

Since RQ1 focussed on identifying and classifying challenges related to product development with OSS components, the researcher decided that a mapping study was the most appropriate research method.

The study had a descriptive nature, and the state of theory can be considered intermediate. RQ1 can be classified as “generalisation or characterisation” (following the classification by Shaw) or in the category “Descriptive and Classification”, following the categorisation by Easterbrook et al. Since this study is a literature review, it can be classified as “DR” (following the classification by Glass et al.).

As was mentioned above, desk research was chosen over field research, as there was a sufficiently large body of literature available. Alternatively, field research could have been conducted in order to address this research question. In particular, survey methodology would have been appropriate to gather insights from a wide range of practitioners. However, this approach, being a so-called “First Order Technique”, is relatively more resource intensive as it involves interaction with practitioners (Lethbridge et al., 2005). Therefore, the researcher decided that desk research was more suitable.

5.5 RESEARCH APPROACH FOR RQ2: ANALYSIS OF OSS EVALUATION METHODS

Research Question 2 is concerned with identifying and reviewing the state-of-the-art of OSS evaluation methods and frameworks. This field of research can be considered to be in an intermediate state (see Table 5-7). While various authors have proposed evaluation methods and frameworks, there has been no systematic identification and comparison of these
methods. The purpose of this study was therefore *descriptive*, and the type of question is “generalisation or characterisation” (following Shaw) and “descriptive-comparative” (following Easterbrook et al.).

In order to address this research question, the researcher identified all relevant literature. The researcher chose to follow the SR guidelines in order to identify all OSS evaluation techniques that have been proposed; however, the literature should not be classified as an SR (nor was it a mapping study). So far, published studies have identified a few OSS evaluation methods and frameworks, but typically limit this discussion to three or four methods.

Rather than using a systematic approach to identify all relevant literature, the researcher could also have chosen to conduct a “traditional” literature review, also discussed briefly in the previous section. However, such an approach could possibly miss relevant papers that are not referenced by others. Also, an ad-hoc approach would not be repeatable.

After identifying all relevant papers, the researcher chose to perform a systematic comparison of the identified OSS evaluation methods. Following other researchers who have performed systematic comparisons, classifications and evaluations of existing methods and processes (e.g., (Matinlassi, 2004; Medvidovic and Taylor, 2000; Ali Babar et al., 2004; Dobrica and Niemelä, 2002)), the researcher chose to design a framework to perform the comparison. One of the purposes of a framework is to help practitioners to “identify strengths, weaknesses and assumption in each classified approach” (Schwarz et al., 2007, p.33).

5.6 RESEARCH APPROACH FOR RQ3: ARCHITECTURAL KNOWLEDGE NEEDS OF OSS INTEGRATORS

Research Question 3 investigates the importance of architectural knowledge of OSS component integrators. The purpose of this study was *exploratory*, and the prior state of theory can be considered to be in its *nascent* phase. The type of question can be considered to be of the type “generalisation and description” (Shaw) or “Description and
Classification” (Easterbrook et al.). As was discussed in Chapter 4, there has been little attention for architectural knowledge in the OSS research field. For these reasons, the researcher decided to conduct a small-scale survey. Easterbrook et al. (2008) write that:

The defining characteristic of survey research is the selection of a representative sample from a well-defined population, and the data analysis techniques used to generalise from that sample to the population, usually to answer base-rate questions.

However, the researcher interprets the term “survey” more generally, following a definition by Pfleeger and Kitchenham (2001), who defined a survey as:

[it] is a comprehensive systems for collecting information to describe, compare or explain knowledge, attitudes and behaviour.

Very similarly, Fink (2003) defined a survey as follows:

A survey is a system for collecting information from or about people to describe, compare, or explain their knowledge, attitudes, and behavior.

Surveys may have various goals; they can be of a descriptive, explanatory, or explorative nature (Babbie, 1990). The aim of a survey is to gather experiences or opinions of a group of individuals. Since little is known about practitioners needs for architectural knowledge, the researcher decided to conduct an explorative survey. As Babbie states, “explorative surveys are used as a pre-study to a more thorough investigation to assure that important issues are not foreseen.” (Babbie, 1990). In this case, a “more thorough” investigation could entail defining a number of hypotheses (based on the results of the explorative survey) that could be tested by means of a more quantitative-oriented survey, using questionnaire as a data collection technique. The goal of this explorative survey was to investigate whether practitioners need architectural knowledge of OSS products.
Fink distinguishes between qualitative and quantitative surveys (Fink, 2003). Fink states that one use for qualitative surveys is,

“To supplement traditional surveys or guide their development in order to:

- Find out which problems and questions are important and should be addressed by the present survey as well as future research and policy;
- Generate research questions and hypotheses;
- […]”

As stated above, this survey’s goal was explorative, which is why the researcher chose to conduct a qualitative survey, rather than a quantitative survey.

The two most common methods to collect data in surveys are questionnaires and interviews (Babbie, 1990). In order to collect data, the researcher chose to conduct interviews with practitioners. Interviews can be structured, unstructured, or semi-structured. The researcher prepared an interview guide enlisting questions that he wanted to investigate, but did not want to limit the interviews to those questions and exclude other topics. For that reason, the researcher chose to conduct semi-structured interviews.

The use of interviews to collect data affects the sample size of the survey. Since interviews are a labour-intensive and costly approach to collect data (Robson, 2002, p.236) (as opposed to using questionnaires, which are less labour-intensive), this limits the number of participants that can be invited to participate in the survey. For this survey, the researcher drew from data gathered from interviews (on average one hour each) with 12 IT practitioners. Participants were selected based on availability; the sample of the study was therefore a convenience sample (Robson, 2002, p.265). Eight interviews were conducted face-to-face, whereas three interviews were telephone interviews, and one interview was conducted using an instant messenger software program. Seven of the interviews were conducted as part of a case study (described in
Section 5.8); during these interviews, data relevant to RQ3 were gathered, which is why the researcher included these interviews in this study.

5.7 RESEARCH APPROACH FOR RQ4: IDENTIFYING ARCHITECTURAL PATTERNS

Research Question 4 was concerned with the design and evaluation of a process for identifying architectural patterns in OSS products. The purpose of this study was to provide guidance to practitioners in this complex task. The overall approach taken to address this research question was based on the “Improvement Paradigm” developed by the Software Engineering Laboratory (SEL) (Parra et al., 1997). Figure 5-5 shows the adapted version of the Improvement Paradigm. The original Improvement Paradigm by Parra et al. focuses on improving a software development process. However, though RQ4 is concerned with improving a process, it is not concerned with a software development process, which is why the model was slightly modified.

The Improvement Paradigm is an iterative process consisting of three steps. The first step is to acquire an understanding of the current situation and develop goals for improving it. This is followed by the second step, which is to assess how to achieve these goals by defining process changes. The third and final step of the cycle is to package the lessons learnt from step two and to integrate these into the “environment”, that is, the current process. Since there are currently no approaches or
processes defined to identify architectural patterns, the “current” process is ad-hoc and undefined.

The researcher performed three iterations of the Improvement Paradigm, in three separate studies. The first iteration (study) focused on acquiring an understanding of approaches and challenges taken by master’s students who had identified architectural patterns in the context of courses on Software Patterns and Software Architecture. Data were collected through eight interviews, 12 experience reports, and the researcher’s personal experience with the pattern identification task. Based on these data, the researcher performed an assessment to determine effective improvements to the task of identifying architectural patterns. In order to package these suggested improvements, the researcher designed the initial version (v1) of a process for Identifying Architectural Patterns in OSS, which was named IDAPO.

The researcher conducted a second iteration; in order to improve the researcher’s understanding, he conducted a second study to gather additional data that could be used to triangulate across data sources (Creswell and Miller, 2000) in order to cross-validate the process’ steps. Data were collected through interviews with another group of 10 master’s students. Based on these data, the researcher assessed how the first version of IDAPO could be improved; these changes were subsequently packaged in an updated version (v2) of IDAPO.

The resulting updated version (v2) of IDAPO was then evaluated in a third iteration; through this evaluation, performed by means of a quasi-experiment involving 14 master’s students, the researcher improved his understanding, based on which a further assessment was performed to identify opportunities for improvement. Though the researcher has performed the assessment that resulted in a number of suggestions for improvement, these are not packaged. This third iteration is therefore not complete, and is left for future work.
5.8 RESEARCH APPROACH FOR RQ5: CHALLENGES IN INNER SOURCE

Research Question 5 investigates challenges and approaches to address these in Inner Source. In Inner Source, an organisation grows a community similar to an OSS community, whereby the community is limited to the confines of the organisation. The purpose of this study was to better understand the challenges that may arise in Inner Source.

The researcher used case study as the research method. Easterbrook et al. (2008) point out that:

There is much confusion in the SE literature over what constitutes a case study. The term is often used to mean a worked example. As an empirical method, a case study is something very different.

The case study as “a worked example” as Easterbrook et al. name it could be categorised as a “Descriptive system” study, following the research approach classification by Glass et al. (see Table 5-1). An example of such a “worked example” case study is reported in (Oručević-Alagić and Höst, 2010).

The researcher adopts Yin’s (2003) interpretation of the term case study as:

an empirical inquiry that investigates a contemporary phenomenon within its real-life context, especially when the boundaries between phenomenon and context are not clearly evident.

In this research, case study approach is justified since the implementation of Inner Source is tailored to the specific characteristics of an organisation (Gaughan et al., 2009). Therefore, the boundaries between the (contemporary) phenomenon (i.e., Inner Source) and context (i.e., the organisation adopting Inner Source) are not clearly evident.

As Verner et al. (2009) point out, case studies may be descriptive, explanatory, exploratory or evaluative. Given the nascent phase of
research in Inner Source, the researcher chose to conduct an *exploratory* case study.

In order to collect data for this case study, the researcher conducted 11 interviews with software developers, software architects and managers. The interviewees worked in different business divisions of a large organisation with a significant track record in the implementation of Inner Source. The interviews were semi-structured, which allowed the researcher to investigate topics that emerged during conversation in further detail.

### 5.9 RESEARCH APPROACH FOR RQ6: WHAT ARE KEY FACTORS TO CONSIDER IN ADOPTING INNER SOURCE?

Research Question 6 investigates the key factors that should be considered in the decision to adopt Inner Source. Chapter 3 reviewed the existing guidelines for adopting Inner Source; this revealed that each of the sets of guidelines addresses one or more aspects, but none provides a holistic overview. In order to address this, the researcher chose to conduct an extensive review of the literature.

Instead of a traditional literature review, the researcher could also have chosen the systematic literature review method to perform the literature review. There were a number of reasons to support the choice of a traditional literature review. Firstly, prior to the research reported in Chapter 11, the researcher had conducted a preliminary (not published) *scoping study* (see Section 5.4) on the topic of Inner Source. Using a search string (in digital libraries provided by IEEE Xplore, ACM Digital Library and SpringerLink, as well as in Google Scholar) that consisted of the various terms used to refer to the use of OSS development practices (see Table 3-3 for an overview), the researcher identified 46 documents of which 20 were relevant to Inner Source, but not necessarily discussing the topic of *adopting* Inner Source. These documents included academic papers and technical reports as well as presentation slides. This limited number indicated that the body of existing literature on the topic of Inner
Source is very limited, and confirmed that the topic as a research field is still very much in its nascent phase. In a survey conducted by Zhang and Ali Babar (2011), a belief that “the study area is relatively small” and/or “insufficient data were available” was given as one justification to not conduct a SLR.

A second reason why the researcher chose a traditional literature review over a SLR is that he was aware of a few relevant documents that were not included in the search results of the preliminary scoping study. In particular, the book “The Cathedral and the Bazaar” by Raymond (2001) was not listed, nor was a book chapter by Robbins (2005).

A third reason is that SLRs are more suitable to summarise existing evidence published in a relatively large number of papers. Since the researcher was looking for existing guidelines rather than evidence, the researcher deemed a narrative literature review more suitable. As stated before, the research area of Inner Source is in its nascent phase, and as such, little evidence is published in peer-reviewed literature. On the other hand, the researcher deemed non-peer reviewed material (such as Raymond’s “The Cathedral and the Bazaar”) quite relevant. In the survey by Zhang and Ali Babar mentioned above, 26% of respondents reported the appropriateness of a narrative review as a reason for not doing a SLR.

Based on the findings of the literature review, the researcher developed a framework that an organisation may use to assess its fit with Inner Source. That is, to what extent is an organisation’s compatible with Inner Source?

In order to demonstrate the framework, the researcher conducted an in-depth industrial case study at an organisation that had indicated an interest in adopting Inner Source. In order to collect data for this case study, the researcher conducted semi-structured interviews with 15 people at the organisation. Most interviewees were software developers. In addition, the researcher studied documentation available within the organisation, and attended a number of meetings. The data were
subsequently analysed using the framework that was developed, as was mentioned above.

5.10 SUMMARY OF THE RESEARCH PROCESS

This chapter presented an overview of the research process that the researcher followed to conduct the research presented in this thesis. The thesis addresses six research questions, for which six studies were conducted. A variety of research methods were used, such as systematic mapping study, survey, case study, and an experiment, and data were collected from different sources. A graphical overview of the research methods and data sources is presented in Figure 5-6.

Figure 5-6. Overview of the research questions, research methods and data collection methods used in this thesis.
The figure shows that 11 interviews were conducted for the case study that the researcher performed to address RQ5. As was mentioned briefly in Section 5.6, data of seven of these interviews were also used to address RQ3, as the researcher had also discussed the topic (of RQ3) with these interviewees.

This chapter has presented an overview and justification of the research method used for each of the six research questions, which are addressed in Chapters 6 to 11 that follow next. These chapters also provide in-depth details about the research designs for each of the studies, as well as discussions of the limitations of each of the studies.
6.1 INTRODUCTION AND CHAPTER LAYOUT

As was discussed in Chapter 2, OSS products are increasingly used in CBSD. The use of OSS components has various benefits, such as significantly lower (purchasing) costs, availability of high-quality products, adherence to open standards, and no vendor dependency. The software development industry has taken note of these benefits, and has been increasingly using OSS components in combination with, or as an alternative to COTS components (Li et al., 2005).

Besides these benefits, several studies have also reported different challenges involved in using OSS components in software development. However, the studies reporting such challenges tend to focus on specific aspects. For example, Merilinna and Matinlassi (2006) reviewed the literature on OSS component integration and compare this with real-world practices. Ven and Mannaert (2008) studied challenges and strategies for ISVs when dealing with modifications and contributions to OSS. Morgan and Finnegan (2007) provide an overview of benefits and drawbacks of adopting OSS. However, their review is mostly based on online articles, such as TechSoup (www.techsoup.org), reports by commercial research institutions (e.g., Forrester) and other types of reports, rather than findings from scientific research literature. Goode

reported a literature review and industrial survey on management barriers to OSS adoption (Goode, 2005).

However, there has been no systematic synthesis of the OSS challenges reported in the literature, which motivated the work reported in this chapter. This chapter presents the findings of a systematic mapping study, which addresses the first research question in this thesis:

RQ1: What are the challenges associated with integration of open source software products?

The researcher asserts that a synthesis of the reported OSS challenges can help practitioners to fully understand the potential OSS challenges and enable them to take appropriate measures to deal with them. Researchers can use the findings for deliberating and debating the possible causes and appropriate strategies for the identified challenges. As part of the overall research strategy discussed in Chapter 5, the researcher chose to identify the challenges associated with integrating OSS products in CBSD following the guidelines for conducting systematic reviews in software engineering proposed by Kitchenham and Charters (2007). However, this study was not a systematic review, but should be characterised as a systematic mapping study; Chapter 5 discussed the differences between the two types of studies. The mapping study was intended to address the following research issues:

- Identify the challenges reported in the literature.
- Rank the identified challenges in order of frequency.
- Identify categories of challenges to provide structure

This chapter presents the detailed design and results of the study, and is organised as follows: Section 6.2 presents the design and administration of the study, followed by the identified challenges in Section 6.3. Limitation of the study are discussed in Section 6.4. The chapter is concluded in Section 6.5.
6.2 RESEARCH METHOD

As mentioned above, the researcher conducted a mapping study to address the research question stated previously. Kitchenham and Charters (2007) present a phased approach to conduct systematic reviews and mapping studies (both of which are from hereon referred to as “review studies”). They distinguish three phases, each of which has a number of stages. These phases and stages are listed in Table 6-1 below.

Table 6-1. Phases and stages for conducting review studies according to Kitchenham and Charters (2007).

<table>
<thead>
<tr>
<th>Phase</th>
<th>Description</th>
<th>Stages</th>
<th>Discussed in</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Planning the review</td>
<td>Identification of the need for a review</td>
<td>Section 6.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Commissioning a review</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Specifying the research question(s)</td>
<td>Section 6.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Developing a review protocol</td>
<td>Section 6.2.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Evaluating the review protocol</td>
<td>Section 6.2.1</td>
</tr>
<tr>
<td>2</td>
<td>Conducting the review</td>
<td>Identification of research</td>
<td>Section 6.2.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Selection of primary studies</td>
<td>Section 6.2.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Study quality assessment</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Data extraction and monitoring</td>
<td>Section 6.2.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Data synthesis</td>
<td>Section 6.2.2</td>
</tr>
<tr>
<td>3</td>
<td>Reporting the review</td>
<td>Specifying dissemination mechanisms</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Formatting the main report</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Evaluating the report</td>
<td>N/A</td>
</tr>
</tbody>
</table>

The column “Discussed in” lists the sections that discuss the various stages. Some stages of Phase 1 have already been discussed in Section 6.1. Not all stages are discussed; for instance, this review was not commissioned by a third party, which is why this stage is not discussed. Furthermore, since this review is a mapping study, the researcher decided that no study quality assessment was required. Lastly, while Kitchenham and Charters discuss the stages in Phase 3 (Reporting the review), these are not discussed in this thesis, since the results are presented in this thesis (as well as in (Stol and Ali Babar, 2010a)).

6.2.1 Development and evaluation of a review protocol

Prior to conducting the review study, the researcher developed a protocol. The protocol contains the research question that the review study aims to answer and the procedure to perform the review study. The search
strategy is discussed in Sub-section 6.2.1.1; this is followed by a discussion of the search string used in this study in Sub-section 6.2.1.2.

6.2.1.1 Search strategy

As the review study is planned, a researcher needs to decide the search strategy. A researcher can choose between two different search strategies (Zhang et al., 2011):

- **Manual search.** In a manual search, the researcher manually selects studies from a predefined number of venues (i.e., journals and conference proceedings). Based on the meta-data (e.g., title, abstract and keywords), and possibly an inspection of the actual publication’s text, the researcher decides whether or not to include the publication in the initial search results.

- **Automated search.** In an automated search, the researcher constructs a “search string”, which is used in a number of digital libraries (e.g., IEEE Xplore and ACM Digital Library), each of which indexes papers published in a number of venues. Each of the libraries yields a number of search results, which together comprise the initial search results.

Both strategies have benefits and drawbacks, in terms of precision and recall of the search results. Precision and recall are standard measures based on a confusion matrix (Olson and Delen, 2008). Table 6-2 presents a confusion matrix. If a paper is relevant and included in the results, it is called a True Positive (TP); if it is not in the results, it is a False Negative (FN). Likewise, if a paper is not relevant but it is included in the results, it is called a False Positive (FP). If it was (correctly) not included in the results, it is a True Negative (TN).

**Table 6-2. Confusion matrix.**

<table>
<thead>
<tr>
<th></th>
<th>Included in results (&quot;positive&quot;)</th>
<th>Not in results (&quot;negative&quot;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relevant</td>
<td>True Positives (TP)</td>
<td>False Negatives (FN)</td>
</tr>
<tr>
<td>Not relevant</td>
<td>False Positives (FP)</td>
<td>True Negatives (TN)</td>
</tr>
</tbody>
</table>

Precision is defined as the fraction of papers correctly identified of the total number of identified papers, i.e., TP ÷ (TP + FP). Recall is defined as the fraction of correctly identified papers of the total number of correct papers, i.e., TP ÷ (TP + FN). Figure 6-1 shows the meaning of precision and recall graphically.
Figure 6-1. Recall and precision depend on the outcome (oval) of a query and its relation to all relevant results (left-hand side of the oval) and the non-relevant results (right-hand side of the oval). (Figure adapted from: Wikipedia\textsuperscript{22}).

Figure 6-1 shows the “search universe” (rectangle) and the query results (oval). Whether or not a result is relevant depends on the precision of the query (horizontal arrow). Whether or not a relevant paper is included in the query results depends on the recall of the query (diagonal arrow).

The set of all relevant papers is called the Gold Standard; as Zhang et al. point out, a perfect search strategy would capture exactly this gold standard (Zhang et al., 2011). It is important, however, to note that it is not possible to establish a gold standard (and therefore, calculating precision and recall), since doing so would require knowledge of the numbers of all four variables in the confusion matrix. However, these concepts are useful in understanding the differences between the two search strategies.

A manual search typically depends on the researcher’s insights, domain expertise and ability to assess a publication’s relevance, which makes it a more subjective approach than an automated search. If the researcher has significant expertise in the topic being studied, the strategy may result in a relatively high precision.

\textsuperscript{22} http://en.wikipedia.org/wiki/Precision_and_recall
However, since this strategy is very time-consuming due to the manual inspection of each publication, a researcher is typically bound by limited resources (time and man power), which is why only a small number of venues can be inspected using this strategy. This will result in a lower recall rate.

An automated search, on the other hand, allows a researcher to search a more extensive range of venues. Searches in digital libraries can be conducted in seconds, resulting in large numbers (e.g., thousands) of results. However, the references of all publications that were identified in the search must be retrieved for further selection steps, which can be an extremely tedious and time-consuming activity. As the search string was finalised in the review protocol, a researcher should not take “shortcuts”, by making early inclusion/exclusion decisions, in order to ensure the study’s rigour.

For this study, the researcher applied a combined search strategy, using both automatic and manual searches. This is discussed in more detail below.

6.2.1.2 Search string
The approach taken by the researcher deviates from the standard procedure as described in (Kitchenham and Charters, 2007). Rather than identifying a set of papers relevant to one particular topic in the OSS literature, the researcher had identified a number of topics of interest to investigate. The researcher decided to take a two-phased approach. In the first phase, the researcher gathered an extensive collection of publications of OSS research. In the second phase, the researcher could then make further selections from this initial collection that are relevant to particular topics. This is illustrated in the Venn diagram in Figure 6-2.
In order to construct a search string that would yield as many publications as possible that have studied an aspect of OSS, the researcher listed all alternative terms and synonyms for “open source software”. This list of terms was as follows:

- “open source software”
- “libre software”
- “free software”
- “OSS”
- “FOSS”
- “F/OSS”
- “F/OSSD”
- “FOSSD”
- “FLOSS”
- “F/LOSS”
- “OSSD”

In order to ensure that no synonyms were left out, the researcher contacted (through email) a colleague researcher who had been a long-term contributor to the Debian Linux project, and was also conducting research in the area. The contacted colleague agreed with the list of terms, and did not identify any other terms that might have been omitted. Therefore, the researcher constructed a search string based on the list of terms that had been identified. The following search string was constructed:
Upon testing the search string, it was found that the search yielded a total of more than 70,000 publications, which the researcher considered to be too much to process. The acronym “OSS” was removed based under the assumption that, if a publication uses this acronym, it is more than likely that the full term “open source software” is also present in the paper. This greatly reduced the number of results, to approximately 10,000 references.

6.2.1.3 Protocol evaluation
After the researcher had developed the protocol, it was evaluated by one other researcher (one of the researcher’s supervisors), before the researcher continued to the next phase, conducting the review. This phase is discussed in the next sub-section.

6.2.2 Conducting the review
This section presents a discussion of the stages of the second phase of the review study, which is “Conducting the review”.

6.2.2.1 Identification and selection of relevant studies
As mentioned in Sub-section 6.2.1.1, the researcher applied both an automatic search and a manual search to identify relevant papers. The researcher started with the automatic search using the search string presented in the previous section. A number of search engines were used, which are listed in Table 6-3. The table also lists the number of identified publications in each search engine.

All references were manually downloaded. A few visiting researchers at the researcher’s institution assisted in this task. All references were stored in Thompson’s EndNote reference management software. After removing duplicates, 10,233 references remained.
Table 6-3. Search engines and results.

<table>
<thead>
<tr>
<th>Search engine</th>
<th>Remarks</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISI Web of Science</td>
<td>-</td>
<td>3,065</td>
</tr>
<tr>
<td>Compendex + Inspec</td>
<td>Only English publications</td>
<td>2,908</td>
</tr>
<tr>
<td>IEEE Xplore</td>
<td>-</td>
<td>885</td>
</tr>
<tr>
<td>SpringerLink</td>
<td>Only looked in “Computer Science” collection</td>
<td>2,427</td>
</tr>
<tr>
<td>ACM Digital Library</td>
<td>-</td>
<td>3,527</td>
</tr>
<tr>
<td>Scirus</td>
<td>Information types = “conferences”; subject area = “Computer Science”</td>
<td>234</td>
</tr>
<tr>
<td>ScienceDirect</td>
<td>title/abstract/keyword; include “journals”; Subjects: “business, management and accounting” + “computer science” + “decision sciences”</td>
<td>167</td>
</tr>
</tbody>
</table>

| Sub-total           | 13,213                                       |
| Total after removing duplicates | 10,233                                       |

After the identification of potentially relevant research papers, the researcher proceeded to select relevant papers. This process is shown in Figure 6-3. The figure shows that the selection took place in two phases, as is also graphically shown in Figure 6-2. The first phase was concerned with selecting papers relevant to open source software. This resulted in a repository of 565 papers. The second phase was concerned with making a sub-selection from this repository of papers that are relevant to product development with OSS. The staged selection procedure is discussed in more detail next.
Stage 1. The identification of relevant studies started with searches in a number of search engines, as summarised in Table 6-3. This resulted in 10,233 titles.

Stage 2. The researcher inspected all of the 10,233 titles, based on which he made an initial selection of 1,258 papers. This selection was only based on paper title. In most cases, it was quite clear whether or not a paper could be excluded. For instance, many papers that were excluded were on the topic of dental flossing (due to the keyword “FLOSS” in the search string).

Stage 3. The researcher inspected each of the 1,258 papers in more detail; in this stage the selection was based on the papers’ abstracts. Based on this inspection, a selection of 565 papers was made. All selected papers
studied an aspect of OSS. This set of 565 papers was a repository from which the researcher could make various sub-selections of papers relevant to a particular aspect of OSS.

**Stage 4.** For the systematic mapping study reported in this chapter, the researcher selected relevant papers from the repository that resulted from Stage 3. The selection was done based on title, and resulted in a selection of 44 papers.

**Stage 5.** In the final stage, the researcher read the full text of all 44 selected papers with the goal to identify and extract relevant data for this mapping study. During this stage, it became clear that a number of papers did not list any challenges to product development with OSS. In some cases, papers discussed the use of Off-The-Shelf components, thereby not differentiating between closed-source and open source components. Since these papers did not contain any relevant data related to challenges in developing with OSS components, these papers were excluded at this stage. This resulted in a final set of 17 papers. These papers are listed in Table 6-4, and are numbered P1 to P17.
<table>
<thead>
<tr>
<th>ID</th>
<th>Title</th>
<th>Author</th>
<th>Year</th>
<th>Published in</th>
</tr>
</thead>
<tbody>
<tr>
<td>P4</td>
<td>Open Source Collaboration for Fostering Off-The-Shelf Components Selection</td>
<td>Ayala, C., Sørensen, C., Conradi, R., Franch, X., and Li, J.</td>
<td>2007</td>
<td>Third International Conference on Open Source Systems</td>
</tr>
<tr>
<td>P5</td>
<td>Why and how to contribute to libre software when you integrate them into an in-house application?</td>
<td>Bac, C., Berger, O., Deborde, V., and Hamel, B.</td>
<td>2005</td>
<td>First International Conference on Open Source Systems</td>
</tr>
<tr>
<td>P6</td>
<td>An empirical study on software development with open source components in the Chinese software industry</td>
<td>Chen, W., Li, J., Ma, J., Conradi, R., Ji, J., and Liu, C.</td>
<td>2008</td>
<td>Software Process: Improvement and Practice</td>
</tr>
<tr>
<td>P8</td>
<td>Surveying Industrial Roles in Open Source Software Development</td>
<td>Hauge, Ø., Sørensen, C.-F., and Rødal, A.</td>
<td>2007</td>
<td>Third International Conference on Open Source Systems</td>
</tr>
<tr>
<td>P9</td>
<td>Experiences on Product Development with Open Source Software</td>
<td>Jaaksi, A.</td>
<td>2007</td>
<td>Third International Conference on Open Source Systems</td>
</tr>
<tr>
<td>P10</td>
<td>The Use of Open Source Software in Enterprise Distributed Computing Environments, Open Source Development, Adoption and Innovation</td>
<td>Krivoruchko, J.</td>
<td>2007</td>
<td>Third International Conference on Open Source Systems</td>
</tr>
<tr>
<td>P13</td>
<td>State of the Art and Practice of Open Source Component Integration</td>
<td>Merilinna, J., and Matinlassi, M.</td>
<td>2006</td>
<td>Euromicro conference on Software Engineering and Advanced Applications</td>
</tr>
<tr>
<td>P14</td>
<td>Using open source software in product development: A primer</td>
<td>Ruffin, C., and Ebert, C.</td>
<td>2004</td>
<td>IEEE software</td>
</tr>
<tr>
<td>P16</td>
<td>Challenges and strategies in the use of Open Source Software by Independent Software Vendors</td>
<td>Ven, K., and Mannaert, H.</td>
<td>2008</td>
<td>Information and Software Technology</td>
</tr>
<tr>
<td>P17</td>
<td>The Importance of External Support in the Adoption of Open Source Server Software</td>
<td>Ven, K., and Verelst, J.</td>
<td>2009</td>
<td>Fifth International Conference on Open Source Systems</td>
</tr>
</tbody>
</table>
6.2.2.2 Publication year and venue of selected studies

Table 6-5 shows the distribution of the included studies by year. The table shows that the studies were published between 2004 and 2009. Most studies included in this review were published in 2005 and 2007.

Table 6-5. Distribution of included studies by year.

<table>
<thead>
<tr>
<th>Year</th>
<th>Number of studies</th>
</tr>
</thead>
<tbody>
<tr>
<td>2004</td>
<td>2</td>
</tr>
<tr>
<td>2005</td>
<td>5</td>
</tr>
<tr>
<td>2006</td>
<td>1</td>
</tr>
<tr>
<td>2007</td>
<td>5</td>
</tr>
<tr>
<td>2008</td>
<td>2</td>
</tr>
<tr>
<td>2009</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 6-6 shows the distribution of included studies by venue and publication type. The table shows that by far most studies were published in the International Conference on Open Source Systems. The remaining studies were published as journal papers (2), magazine articles (2), a paper in a general conference (Euromicro’s SEAA track) and an OSS workshop paper.

Table 6-6. Distribution of included studies by venue.

<table>
<thead>
<tr>
<th>Venue</th>
<th>Type</th>
<th>Number of studies</th>
</tr>
</thead>
<tbody>
<tr>
<td>International Conference on Open Source Systems</td>
<td>Conference</td>
<td>11</td>
</tr>
<tr>
<td>Software Process: Improvement and Practice</td>
<td>Journal</td>
<td>1</td>
</tr>
<tr>
<td>IEEE Software</td>
<td>Magazine</td>
<td>2</td>
</tr>
<tr>
<td>Information and Software Technology</td>
<td>Journal</td>
<td>1</td>
</tr>
<tr>
<td>Workshop on Open Source Software Engineering</td>
<td>Workshop</td>
<td>1</td>
</tr>
<tr>
<td>Euromicro conference on Software Engineering and Advanced Applications (SEAA)</td>
<td>Conference</td>
<td>1</td>
</tr>
</tbody>
</table>

6.2.2.3 Data extraction

The researcher thoroughly read all identified papers to identify any reported challenges related to using OSS in product development. Some papers explicitly listed challenges, whereas others implicitly reported the issues. The researcher extracted challenges that were either reported as authors’ experiences, or as cited from other literature. While reading the papers, all challenges were recorded in a spreadsheet document. After
extracting the challenges, each one of them was annotated with one or more keywords. In particular, a number of challenges were found that could be tagged as ‘community’, ‘support’ and ‘maintenance’, since maintenance (e.g., bug fixes) is one type of support, which is provided by the community. Based on these keywords, a number of categories emerged, which were used to cluster related challenges. During analysis of the challenges, similar challenges that essentially stated the same issue were merged.

6.3 CHALLENGES IN INTEGRATING OPEN SOURCE SOFTWARE

The results of the review are presented in Table 6-7. Twenty-one challenges were identified that have been reported in literature, divided into the six categories that emerged during the analysis of the extracted data from the reviewed papers. The numbers after the categories’ names indicate the total number of papers reporting challenges in that category; for instance, nine papers reported challenges related to Product Selection.

The remainder of this section discusses each of these challenges. Table 6-7 provides references (using the identifiers P1 to P17) to the papers listed in Table 6-4 that have reported the challenges. The challenges are numbered C1 to C21; these identifiers are used throughout the remainder of this section.
Table 6-7. Challenges in integrating OSS components in product development.

<table>
<thead>
<tr>
<th>Category</th>
<th>ID</th>
<th>Challenge</th>
<th>Freq.</th>
<th>Reported in</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Product Selection (9)</strong></td>
<td>C1</td>
<td>Identifying quality products among the large supply is difficult due to uncertainty about quality (e.g. usability, stability, reliability)</td>
<td>7</td>
<td>P8, P3, P13, P6, P7, P10, P15</td>
</tr>
<tr>
<td></td>
<td>C2</td>
<td>Lack of time to evaluate components</td>
<td>1</td>
<td>P3</td>
</tr>
<tr>
<td></td>
<td>C3</td>
<td>Decide what “fork” of the project should be chosen</td>
<td>1</td>
<td>P5</td>
</tr>
<tr>
<td><strong>Documentation (5)</strong></td>
<td>C4</td>
<td>Lack of, or low quality documentation</td>
<td>3</td>
<td>P3, P11, P13</td>
</tr>
<tr>
<td></td>
<td>C5</td>
<td>Several descriptions of the same component</td>
<td>1</td>
<td>P4</td>
</tr>
<tr>
<td><strong>Community, support and maintenance (19)</strong></td>
<td>C6</td>
<td>Dependency on the community for further support and upgrades; possible need to hire additional talent for maintenance; difficult to control the quality of the support; lack of helpdesk and technical support.</td>
<td>5</td>
<td>P7, P6, P10, P13, P15</td>
</tr>
<tr>
<td></td>
<td>C7</td>
<td>Custom changes need to be maintained, which is time-consuming and may cause problems with future versions/community may take a different, incompatible approach.</td>
<td>6</td>
<td>P8, P5, P9, P17, P12, P16</td>
</tr>
<tr>
<td></td>
<td>C8</td>
<td>Convincing OSS community to accept changes (modifications may be too specific); contributions can be difficult or costly. Difficult to control the architecture if not a core member.</td>
<td>5</td>
<td>P8, P5, P13, P12, P16</td>
</tr>
<tr>
<td></td>
<td>C9</td>
<td>Uncertainty about product future and consequences for company product</td>
<td>1</td>
<td>P5</td>
</tr>
<tr>
<td></td>
<td>C10</td>
<td>Community members would like to have a bigger say in features and integrating final product with company</td>
<td>1</td>
<td>P9</td>
</tr>
<tr>
<td></td>
<td>C11</td>
<td>Contributing and investing in OSS project costs resources</td>
<td>1</td>
<td>P9</td>
</tr>
<tr>
<td><strong>Integration and Architecture (8)</strong></td>
<td>C12</td>
<td>Backward compatibility concerns</td>
<td>2</td>
<td>P9, P15</td>
</tr>
<tr>
<td></td>
<td>C13</td>
<td>Modifications needed to implement missing functionality or fit into architecture</td>
<td>2</td>
<td>P15, P16</td>
</tr>
<tr>
<td></td>
<td>C14</td>
<td>Incompatibility between components or existing systems</td>
<td>2</td>
<td>P7, P15</td>
</tr>
<tr>
<td></td>
<td>C15</td>
<td>Horizontal integration</td>
<td>1</td>
<td>P13</td>
</tr>
<tr>
<td></td>
<td>C16</td>
<td>Vertical integration / Mismatch of platform/programming language</td>
<td>1</td>
<td>P13</td>
</tr>
<tr>
<td><strong>Migration and usage (3)</strong></td>
<td>C17</td>
<td>Complexity of configuration</td>
<td>1</td>
<td>P15</td>
</tr>
<tr>
<td></td>
<td>C18</td>
<td>User training/learning costs</td>
<td>2</td>
<td>P7, P15</td>
</tr>
<tr>
<td><strong>Legal &amp; Business (3)</strong></td>
<td>C19</td>
<td>Complex licensing situation</td>
<td>5</td>
<td>P14, P11, P9, P1, P15</td>
</tr>
<tr>
<td></td>
<td>C20</td>
<td>Concerns about, or no clear strategy on Intellectual Property Rights (IPR) issues</td>
<td>3</td>
<td>P14, P1, P15</td>
</tr>
<tr>
<td></td>
<td>C21</td>
<td>Lack of clear business models that are appealing to industry</td>
<td>2</td>
<td>P1, P7</td>
</tr>
</tbody>
</table>
6.3.1 Product selection

Three challenges were identified that are related to the topic of product selection. These are: (1) too much choice, (2) lack of time to evaluate, and (3) choosing a “fork” of the product. These are discussed next.

6.3.1.1 Too much choice

Various studies have reported that identifying quality products among the many available OSS products is difficult due to an uncertainty about the quality (challenge C1). The quality is typically referred to in terms of quality attributes such as usability, reliability and performance. Sourceforge.net, the largest repository for OSS projects, alone hosts more than 260,000 projects\(^{23}\). This challenge has long been recognised, and has resulted in a variety of OSS evaluation methods and frameworks, such as Capgemini’s Open Source Maturity Model (OSMM) (Duijnhouwer and Widdows, 2003), Navica’s OSMM (Golden, 2004), OpenBQR (Taibi et al., 2007) and QSOS (Atos Origin, 2006). Chapter 7 provides an extensive overview of these methods.

However, despite these efforts, research has shown that practitioners typically do not use these evaluation methods and frameworks (Li et al., 2009; Merilinna and Matinlassi, 2006). Rather, they use ad-hoc approaches and information sources to select components, such as experiences of colleagues. Hauge et al. found that a “first fit” rather than a “best fit” principle is applied (Hauge et al., 2009).

6.3.1.2 Lack of time to evaluate

A related challenge to C1 is a lack of time to evaluate components (C2). Though this is of course a direct consequence of having a large number of OSS products to evaluate, the researcher decided that it should be listed separately, since it was separately identified by a study (Ayala et al., 2009) that also reported challenge C1. Furthermore, possible measures that practitioners could take to address C2 are different to measures to address C1 and distinguish C1 more clearly from C2. Such measures

\(^{23}\)http://sourceforge.net/about
include: (1) limiting the number of components to evaluate to a small number, and (2) a decision on management level to allocate more resources (time and manpower) for evaluating OSS components.

6.3.1.3 Choosing a fork
Another related challenge to C1 is deciding what “fork” of the OSS project should be used (C3). If a fork occurs, a new project is spun off from the original project, and can occur if a project’s core developers have fundamental disagreements about the future of the project (Spinellis and Szyperski, 2004). This challenge is different from C1, since C1 refers to making the decision on what product to select, whereas C3 refers to what fork of that product should be selected. Forking projects rarely happens, and there is a strong social pressure against forking (Raymond, 2001). Nevertheless, if it happens, developers need to decide which fork of the project to select. One experience paper reported that this decision caused a temporary delay in development (Bac et al., 2005).

6.3.2 Documentation
Unsurprisingly, documentation has emerged as a category of challenges. Two challenges were identified: (1) lack of quality documentation, and (2) different descriptions of the same component. These are discussed next.

6.3.2.1 Lack of quality documentation
A lack of good quality documentation remains to be a challenge that is difficult to overcome (C4). Well-documented software is easier to understand by others, which makes it easier to modify the software. However, OSS contributors are typically more interested in coding, and some consider adding comments in the source code is sufficient (Gacek and Arief, 2004). Researchers have proposed various architecture recovery methods to overcome the lack of design and architecture documentation (Ducasse and Pollet, 2009), as was discussed in Chapter 4.
6.3.2.2 Different descriptions of the same component

The availability of different descriptions of the same components is problematic as well (C5). OSS products may have documentation, but due to the active evolution of many OSS products, this documentation may quickly go out of date.

6.3.3 Community, support and maintenance

A wide variety of challenges have been reported with respect to the interaction with the community. This interaction can be related to (future) support for the product as well as contributions to the project. Maintenance can be done by both the OSS product’s community as well as through contributions from the product’s users. In fact, the boundary between “community members” and “users” may not be that clear according to the onion model (Crowston and Howison, 2005), which states that the social structure of an OSS community is layered, and users are just another layer (see Section 2.4.1). As Gacek and Arief state: “all OSS developers are users, but not all users are developers” (Gacek and Arief, 2004).

6.3.3.1 Dependency for future support

If an organisation decides to use an OSS product, it is dependent on the community for future support and upgrades. A challenge is to acquire support for the OSS product that is of sufficient quality (C6). Support for an OSS product is provided on a voluntary basis by the community, which makes it difficult to control the level of quality of support that is needed. However, for some products, there is an option to acquire support and training from companies such as Red Hat and IBM (Feller and Fitzgerald, 2002). Various studies report the support to be a challenge (see Table 6-7). Ven and Verelst reported a study that investigated the reliance of organisations on commercial support (Ven and Verelst, 2009). They found that the absence of available commercial support is not an insurmountable obstacle for adopting OSS. However, they also found that the OSS community is primarily used by organisations with a strong technical background.
6.3.3.2 **Need to maintain custom changes**

If changes are made to an OSS product, and these modifications are not given back to the product’s community, then the software developers that made such changes need to maintain these custom changes themselves (C7). This means that additional resources must be allocated to the maintenance efforts. Furthermore, if an organisation does not give back such modifications, it effectively *forks* the project as the customised modifications define a new version of that product. This may have serious consequences for future compatibility. Modifications may have to be re-applied whenever new versions of the OSS are used. As an OSS product evolves, patches that implement modifications may no longer be easily applied. Alternatively, an OSS product’s community may tackle a certain feature or issue themselves by taking an approach that may be incompatible with the customised modifications (Ven and Mannaert, 2008).

6.3.3.3 **Difficult to get changes accepted**

An organisation may decide to contribute the changes made back to an OSS component (that is integrated into a product). However, sending patches to an OSS’s community does not automatically imply that these patches get accepted (C8). A key characteristic of an OSS development process is that contributions are thoroughly scrutinised by community members that have commit access (Feller and Fitzgerald, 2002). In general, any changes or proposals for change will be subject to a review process (Raymond, 2001). Furthermore, an OSS project may have specific practices that developers will have to get familiar with (Bac et al., 2005). Specific extensions may be rejected to prevent too many new features (code bloat) being introduced by one-time contributors (Ven and Mannaert, 2008).

6.3.3.4 **Uncertainty about product future**

A realistic concern that organisations may have is the future, or *longevity* of the OSS product (C9). Obviously, if a certain OSS product is adopted, an adopter does not like to be in a situation where the community supporting that product disappears. If that happens, it means no future
support or updates for that particular product will be available. In such a case, an organisation may choose to take over the maintenance of the project. However, this would result in additional maintenance efforts, and may distract the organisation from its core business.

6.3.3.5 Community wants more influence
One study reporting experiences of product development with OSS at Nokia reported a challenge similar to challenge C8, but in the opposite direction. In this case, OSS developers expressed their wish to be more closely involved in features of the final product. The closed way of integrating these OSS products can cause frustration among OSS developers (Jaaksi, 2007) (C10). To partially address this issue, Nokia started a special distribution, to allow anybody participate more closely in the development. However, as stated in (Jaaksi, 2007), product companies must have the final control over their products.

6.3.3.6 Contributing costs resources
An organisation may choose to use OSS as-is without further development. Alternatively, using OSS becomes more effective if the organisation actively participates in a community’s development process. This, however, requires additional resources (C11). The amount of resources required depends on the level of involvement. Bonaccorsi et al. (2007) list three kinds of involvement: (1) project coordination; (2) code development collaboration, and (3) provision of code.

6.3.4 Integration and architecture
Five challenges were identified related to the integration of OSS components, which is also related to a component’s architecture. These are discussed next.

6.3.4.1 Backward compatibility issues
An OSS product is continuously evolving, depending on the liveliness of a community. Changes to products include new features, bug fixes and architectural changes. After an organisation starts using an actively evolving product, new versions are released. As a product’s development
continues, at some point newer versions are no longer backward compatible, which can become a problem if the product in which the OSS is integrated depends on certain features or APIs (C12).

An organisation will have to adopt a strategy for updating any used OSS components. On the one hand, an organisation may choose to stay close to the latest version of the OSS component. However, this has consequences for backward compatibility, as features may be deprecated and architectural changes may occur (Jaaksi, 2007). Ven and Mannaert describe four possible strategies for contributing to OSS projects (Ven and Mannaert, 2008): (1) contributing any modifications, (2) taking regular snapshots, (3) forking and (4) initiating an OSS project as a set of patches to an existing OSS project. One solution to this problem is to use only those distributions that are provided by packaging companies (Ruffin and Ebert, 2004).

6.3.4.2 Need for modifications
A consequence of a CBSD approach is that components must be fitted into a system. That means the components may have to be modified. Furthermore, OSS components may have to be modified if they do not have all required functionality. Such modifications require additional resources (C13). Obviously, OSS components are more flexible than COTS because the source code of OSS components is available for modifications. However, many organisations do not usually make any changes to the source code before using OSS components (Li et al., 2009).

6.3.4.3 Component and architecture incompatibilities
OSS components may not be compatible with each other, or with existing architectures (C14). This phenomenon is called architectural mismatch, which may have serious consequences for the development schedule and costs (Garlan et al., 1995). Another compatibility issue that may arise is that components may have dependencies on conflicting libraries (Ven and Mannaert, 2008).
6.3.4.4 **Horizontal integration issues**

Merilinna and Matinlassi (2006) distinguish horizontal integration issues at the architectural level and the component level (C15). No specific architectural level issues have been identified. For the component level, four design level contracts have been discussed in (Beugnard *et al.*, 1999). These levels are: (1) syntactic interface, (2) (pre- and post-conditional) constraints, (3) synchronization and timing and (4) quality-of-service. Each of these levels can have associated challenges.

6.3.4.5 **Vertical integration issues**

A mismatch of platform and programming language is an example of vertical integration issues (C16). Platforms may be hardware (i.e., processor types) or software (operating systems and virtual machines (VM) such as the Java VM, .NET and Parrot VM). Some of the techniques to overcome the vertical integration problems are use of middleware (e.g., CORBA), virtual machines (e.g., JVM, a hardware platform-independent virtual machine) and Model-Driven Architecture (MDA) (Merilinna and Matinlassi, 2006).

6.3.5 **Migration and usage**

Complexity of configuring or setting up a user-environment can be an issue (C17). One study reported that significant effort was required to set up an installation of software (approximately three weeks) (Tiangco *et al.*, 2005).

Two studies reported additional cost involved in migration to an OSS alternative and staff training to be a challenge (C18). In this study, the researcher focusses on product development with OSS rather than adopting OSS in favour of proprietary solutions. Migration costs seem to imply the migration from a proprietary solution to an OSS solution, such as the migration from Microsoft Office to OpenOffice.org. However, such end-user applications may be integrated as a sub-system of a larger solution.
6.3.6 Legal and business

6.3.6.1 Complex licensing situation
Not surprisingly, several studies have reported the complex OSS licensing situation to be a challenge (C19). One study reports a lack of consistency between licensing agreements and little guidance on interpreting the Open Source licenses (Tiangco et al., 2005). At the time of writing, the Open Source Initiative lists 69 licenses that comply with the “Open Source Definition” (Open Source Initiative, 2011). It is therefore not a surprise that OSS licensing is perceived to be a complex issue. Some research efforts have been made to address this issue. Alspaugh et al. (2009) present a license analysis scheme, and an approach to automatically analyse license interactions. German and González-Barahona (2009) have documented a number of strategies that developers have used to legally circumvent some restrictions of the GPL.

6.3.6.2 Concerns and issues regarding IPR
Organisations that use OSS products may have concerns about Intellectual Property Rights (IPR) (C20). Source code may have been illegally used in an OSS product and propagated. For instance, in recent years there have been some claims from Microsoft saying that Linux uses their intellectual property (TheRegister, 2006). In that particular case, a deal was made with Novell, so that customers of Novell’s SUSE Linux distribution are protected from any claims.

6.3.6.3 Lack of clear business models
Two studies reported a lack of clear business models for using OSS (C21). It is worthwhile to note that both studies were published in 2005; however, this issue has not been mentioned in more recent literature. In (Gacek and Arief, 2004), three business models have been identified that motivate organisations to get involved in OSS: (1) software for own use, (2) packaging and selling of the software and (3) a platform for commercial or research software development.
6.4 LIMITATIONS OF THIS STUDY

The researcher is aware of a few limitations of this mapping study presented in this chapter, which are discussed in this section.

Firstly, though the researcher performed a rigorous literature search, studies may have unintentionally been excluded due to the subjectivity of the inclusion and exclusion criteria. This is a common limitation in systematic reviews of the literature (which include mapping studies); rather than including any seemingly interesting study that a researcher finds, a researcher should follow the study protocol in order to limit a researcher’s bias in the selection phase, thereby making the study process repeatable and transparent.

Secondly, the selection of the papers was performed solely by the researcher, and was not crosschecked by others. The researcher interpreted the selection criteria liberally in the initial stage, in order to make sure that “borderline” papers (i.e., papers of which the researcher was not sure to include them) are thoroughly inspected for relevancy in a later stage of the selection process.

Thirdly, the classification of challenges is necessarily subjective. However, it is not the researcher’s intention to present a definitive classification; rather, the researcher intended to present the findings in a structured way that can help practitioners to inform them of challenges in using OSS in product development that have been reported so far, and may therefore arise in their situation

6.5 CHAPTER SUMMARY

This section summarises the findings from the systematic mapping study of the literature on product development with OSS components. Through a systematic identification of relevant literature, 21 different challenges were identified related to product development with OSS components. The challenges have been classified into six different categories:
Product selection (3 challenges);
Documentation (2 challenges);
Community, support and maintenance (6 challenges);
Integration and architecture (5 challenges);
Migration and usage (2 challenges);
Legal and business (3 challenges).

Identifying quality products among the large supply of OSS products was the most frequently reported challenge (C1). Other challenges that were reported frequently were:

- Lack of, or low quality documentation (C4);
- Dependency on community for future support (C6);
- Need for maintenance of custom changes (C7);
- Getting changes accepted by the OSS project’s community (C8);
- Complex licensing situation (C19).

Product selection is an important aspect in CBSD and has long been recognised to be a challenge. As a result, a variety of OSS evaluation methods, frameworks and techniques have been proposed. Chapter 7 which follows next presents a framework-based analysis and comparison of these methods, in order to identify to what extent these methods support OSS integrators.
Chapter 7

An Analysis of Evaluation Methods for Open Source Software*

7.1 INTRODUCTION AND CHAPTER LAYOUT

Chapter 6 presented an overview of challenges that have been reported in the literature that practitioners may face in Component-Based Software Development (CBSD) with OSS components. One of the most frequently reported challenges is the selection of suitable OSS components. Both researchers and practitioners have attempted to address this issue, and have proposed a variety of OSS evaluation and selection methods.

However, research has shown that practitioners rarely use formal selection procedures (Li et al., 2009). Instead, OSS products are often selected based on familiarity or recommendations by colleagues (Hauge et al., 2009). For practitioners it may be difficult to choose a suitable evaluation method. The lack of adoption of these evaluation approaches by practitioners may be a result of a deficiency in clarity of the OSS evaluation methods landscape. There has been no systematic comparison of the existing OSS evaluation methods. David A. Wheeler lists a number of evaluation methods in (Wheeler, 2009), but does not provide a thorough comparison of existing evaluation methods. The researcher is aware of two papers that present comparisons of a few methods. Deprez and Alexandre (2008) provide an in-depth comparison of two methods, namely Qualification and Selection of Open Source Software (QSOS) and

Business Readiness Rating for Open Source (commonly abbreviated to OpenBRR). However, it is not feasible to extend their approach to compare a large number of methods. Petrinja et al. (2010) performed a comparison of three assessment models for open source, namely: OpenBRR, QSOS and the OpenSource Maturity Model (OMM). This comparison was performed through a controlled experiment using undergraduate and master’s students.

In order to gain insights into what guidance exists currently to support practitioners in the task of selecting suitable OSS products, this chapter presents a systematic identification and comparison of OSS evaluation methods. The chapter is structured as follows. Section 7.2 discusses the procedure followed to systematically identify OSS evaluation methods. Section 7.3 presents and justifies a Framework for Comparing Open Source software Evaluation Methods (FOCOSEM). The results of the comparison based on FOCOSEM are presented in Section 7.4. Section 7.5 summarises this chapter.

### 7.2 IDENTIFICATION OF OSS EVALUATION METHODS

For the identification of the various OSS evaluation methods, four different sources were consulted. Firstly, all papers in the repository resulting from Phase 1 of the systematic review outlined in Figure 6-3 were screened to identify any OSS evaluation technique or method. All papers reporting a method, framework or any other proposed way of evaluating an OSS product were included. Papers presenting an approach for selecting COTS (as opposed to an exclusive focus on OSS components) were excluded. Secondly, the researcher inspected the “related work” sections of the selected papers. The researcher also noticed that a number of OSS evaluation methods were not reported in research publications, but only appeared in books or white papers. Since those methods were often referenced in the “related work” sections of many papers, the researcher decided to include those methods in this research. Thirdly, the researcher manually selected publications reported in the proceedings of the five International Conferences on Open Source
Lastly, the researcher used his knowledge of the field in order to identify some approaches. It is worthwhile to note that the researcher deliberately did not consider any websites (such as web logs) presenting pragmatic “tips for selecting OSS”. The researcher made this choice based on two grounds. Firstly, while individual web logs with such “tips” may provide some useful input, the quality of the web logs varies widely. Some “bloggers”, such as Wheeler (2009), provide rather extensive discussions of the topic, whereas other web logs do not. In order to ensure a certain quality (or “thoroughness”) of proposed approaches, the researcher believes it is justified to limit inclusion of material presented in books and academic literature.

A second reason to support the decision of excluding weblogs is that web logs are not permanent. It is generally not uncommon that weblogs or websites are abandoned. Including such sources could therefore lead to results that may not be inspected after a weblog or site becomes unavailable.

Following the abovementioned search process, 20 approaches were identified for OSS evaluation. Table 7-1 lists the identified OSS evaluation approaches in chronological order of publication. The column “Source” lists references to papers and reports that reported the method, and can be used by interested readers for further investigation. The column “Orig.” indicates whether the initiative came from industry (I) or from a research (R) setting. A method was considered to be an industry initiative if it was associated with a company name; otherwise it was considered to be a researchers’ initiative. The column “Method” indicates whether it is a well-defined method outlining the required activities, tasks, inputs, and outputs, as opposed to a mere set of evaluation criteria. As can be seen from Table 7-1, only half of the approaches that were identified are methods.

Methods that are investigated in more detail later in this chapter are marked with an asterisk (“*”) in Table 7-1. Given the limited resources of the researcher, a selection of six methods was made for the
comparison. Methods were mainly selected based on the criterion whether or not they were cited, as this suggests that a method is better known than not-cited methods (and therefore are more likely to be introduced to practitioners). Another reason for selecting methods is if they are part of a larger research project; a number of methods are proposed in the context of national or European research; such method proposals are more likely to be followed up in future work in the context of such projects. The methods OpenBQR and OpenBRR are closely related, and both often cited; the researcher decided to include only one of these due to their expected similarities. The researcher also tried to select methods published in different years.

The comparison is presented in Section 7.4. The next section presents the comparison framework that is used for that purpose.
Table 7-1. OSS evaluation methods and frameworks that have been proposed by researchers and industry.

<table>
<thead>
<tr>
<th>No.</th>
<th>Name</th>
<th>Year</th>
<th>Reported in</th>
<th>Origin</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Capgemini Open Source Maturity Model (Capgemini OSMM)</td>
<td>2003</td>
<td>(Duijnhouwer and Widdows, 2003)</td>
<td>Industry</td>
<td>Yes*</td>
</tr>
<tr>
<td>3</td>
<td>A Model for Comparative Assessment of Open Source Products</td>
<td>2004</td>
<td>(Polančič and Horvat, 2004; Polančič et al., 2004)</td>
<td>Research</td>
<td>Yes</td>
</tr>
<tr>
<td>4</td>
<td>Navica Open Source Maturity Model (Navica OSMM)</td>
<td>2004</td>
<td>(Golden, 2004)</td>
<td>Industry</td>
<td>Yes*</td>
</tr>
<tr>
<td>5</td>
<td>Woods and Guliani’s OSMM</td>
<td>2005</td>
<td>(Woods and Guliani, 2005)</td>
<td>Industry</td>
<td>No</td>
</tr>
<tr>
<td>6</td>
<td>Open Business Readiness Rating (OpenBRR)</td>
<td>2005</td>
<td>(Wasserman et al., 2006; <a href="http://www.openbrr.org">www.openbrr.org</a>, 2005)</td>
<td>Research/Industry</td>
<td>Yes</td>
</tr>
<tr>
<td>8</td>
<td>Evaluation Criteria for Free/Open Source Software Products</td>
<td>2006</td>
<td>(Cruz et al., 2006)</td>
<td>Research</td>
<td>No</td>
</tr>
<tr>
<td>9</td>
<td>A Quality Model for Open Source Software Selection</td>
<td>2007</td>
<td>(Sung et al., 2007)</td>
<td>Research</td>
<td>No</td>
</tr>
<tr>
<td>10</td>
<td>Selection Process of Open Source Software</td>
<td>2007</td>
<td>(Lee et al., 2007)</td>
<td>Research</td>
<td>Yes</td>
</tr>
<tr>
<td>11</td>
<td>Observatory for Innovation and Technological transfer on Open Source software (OITOS)</td>
<td>2007</td>
<td>(Cabano et al., 2007)</td>
<td>Research</td>
<td>Yes*</td>
</tr>
<tr>
<td>12</td>
<td>Framework for OS Critical Systems Evaluation (FOCSE)</td>
<td>2007</td>
<td>(Ardagna et al., 2007)</td>
<td>Research</td>
<td>No</td>
</tr>
<tr>
<td>13</td>
<td>Balanced Scorecards for OSS</td>
<td>2007</td>
<td>(Lavazza, 2007)</td>
<td>Research</td>
<td>No</td>
</tr>
<tr>
<td>14</td>
<td>Open Business Quality Rating (OpenBQR)</td>
<td>2007</td>
<td>(Taibi et al., 2007)</td>
<td>Research</td>
<td>Yes*</td>
</tr>
<tr>
<td>15</td>
<td>Evaluating OSS through Prototyping</td>
<td>2007</td>
<td>(Carbon et al., 2007)</td>
<td>Research</td>
<td>Yes</td>
</tr>
<tr>
<td>16</td>
<td>A Comprehensive Approach for Assessing Open Source Projects</td>
<td>2008</td>
<td>(Ciolkowski and Soto, 2008)</td>
<td>Research</td>
<td>No</td>
</tr>
<tr>
<td>17</td>
<td>Software Quality Observatory for Open Source Software (SQO-OSS)</td>
<td>2008</td>
<td>(Samoladas et al., 2008)</td>
<td>Research</td>
<td>Yes*</td>
</tr>
<tr>
<td>18</td>
<td>An operational approach for selecting open source components in a software development project</td>
<td>2008</td>
<td>(Majchrowski and Deprez, 2008)</td>
<td>Research</td>
<td>No</td>
</tr>
<tr>
<td>19</td>
<td>QualiPSo trustworthiness model</td>
<td>2008</td>
<td>(del Bianco et al., 2009; del Bianco et al., 2008)</td>
<td>Research</td>
<td>No</td>
</tr>
<tr>
<td>20</td>
<td>OpenSource Maturity Model (OMM)</td>
<td>2009</td>
<td>(Petrinja et al., 2009)</td>
<td>Research</td>
<td>No</td>
</tr>
</tbody>
</table>
7.3 A COMPARISON FRAMEWORK FOR OPEN SOURCE SOFTWARE EVALUATION METHODS

In order to perform a systematic comparison of the selected OSS evaluation methods, the researcher designed a comparison framework called Framework fOr Comparing Open Source software Evaluation Methods (FOCOSEM), which is presented in Table 7-2. One of the purposes of a framework is to “identify strengths, weaknesses and assumptions in each classified approach” (Schwarz et al., 2007, p.33).

Table 7-2. Framework for Comparing Open Source software Evaluation Methods (FOCOSEM).

<table>
<thead>
<tr>
<th>Component</th>
<th>Element</th>
<th>Brief description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method Context</td>
<td>Specific goal</td>
<td>What is the particular goal of the method?</td>
</tr>
<tr>
<td>Method Context</td>
<td>Functionality evaluation</td>
<td>Is functionality compliance part of the evaluation method?</td>
</tr>
<tr>
<td>Method Context</td>
<td>Results publicly available</td>
<td>Are evaluations of OSS products stored in a publicly accessible repository?</td>
</tr>
<tr>
<td>Method Context</td>
<td>Relation to other methods</td>
<td>How does the method relate to other methods? I.e. what methods was this method based on?</td>
</tr>
<tr>
<td>Method User</td>
<td>Required skills</td>
<td>What skills does the user need to use the method?</td>
</tr>
<tr>
<td>Method User</td>
<td>Intended users</td>
<td>Who are the intended users of the method?</td>
</tr>
<tr>
<td>Method Process</td>
<td>Method’s activities</td>
<td>What are the evaluation method’s activities and steps?</td>
</tr>
<tr>
<td>Method Process</td>
<td>Number of criteria</td>
<td>How many criteria are used in the evaluation?</td>
</tr>
<tr>
<td>Method Process</td>
<td>Evaluation categories</td>
<td>What are the method’s categories of criteria based on which the OSS product is evaluated?</td>
</tr>
<tr>
<td>Method Process</td>
<td>Output</td>
<td>What are the outputs of the evaluation method?</td>
</tr>
<tr>
<td>Method Process</td>
<td>Tool support</td>
<td>Is the evaluation method supported by a tool?</td>
</tr>
<tr>
<td>Method Evaluation</td>
<td>Validation</td>
<td>Has the evaluation method been validated?</td>
</tr>
<tr>
<td>Method Evaluation</td>
<td>Maturity stage</td>
<td>What is the maturity stage of the evaluation method?</td>
</tr>
</tbody>
</table>

FOCOSEM is based on four different sources to justify the selection and formation of its components and elements. The first source is the NIMSAD framework, which is a general framework for understanding and evaluating any methodology (Jayaratna, 1994). NIMSAD, short for Normative Information Model-based System Analysis and Design, defines four components to evaluate a methodology: (1) the problem situation (“context”), (2) the problem solver (“user”), (3) the problem-solving process (“process”), and (4) the method’s evaluation (“evaluation”). Previously, NIMSAD has been used as a basis for the
development of a number of other comparison frameworks in software engineering (Matinlassi, 2004; Forsell et al., 1999; Ali Babar and Gorton, 2004). The second source for FOCOSEM is FOCSAAM, which is a comparison framework for software architecture analysis methods (Ali Babar et al., 2004). The third source is a comparison framework for software product line architecture design methods (Matinlassi, 2004). As a fourth source, the researcher identified differences and commonalities among various OSS evaluation methods. It is noted that the objective of this structural comparison is not to make any judgments about different OSS evaluation methods, although the framework-based comparison can be used to identify weaknesses and strengths of each method (Schwarz et al., 2007). Instead, this study presents an overview of the OSS evaluation method landscape. Furthermore, the insights gained through the framework-based comparison may help practitioners to select a suitable OSS evaluation method. The remainder of this section discusses and justifies the comparison criteria in more detail.

7.3.1 Method Context

The first component, “method context”, contains comparison elements that characterise the context, or situation, in which a method is to be used. The remainder of this sub-section describes the elements of the “method context” component in more detail.

7.3.1.1 Method’s specific goal

A new OSS evaluation method often comes into existence when researchers or practitioners find that existing methods lack certain features required for a particular kind of evaluation or do not meet certain evaluation goals. Although all OSS evaluation methods share the same general objective, namely the evaluation of the suitability of an OSS product, each method has a different approach. The knowledge of a method’s goal is expected to help select the right method for the kind of evaluation to be performed.
7.3.1.2 Functionality evaluation

Some of the identified OSS evaluation methods focus only on the quality of the OSS product, whereas others also evaluate the functionality. This is an essential distinction, since it indicates an evaluator’s flexibility regarding the functionality of OSS products. A method that considers functionality as a criterion assumes that there is a certain level of flexibility regarding the required functionality of the product to be selected. If it is not a selection criterion, the assumption is that the functionality of the product being evaluated conforms to the requirements. This affects whether or not a method is suitable to be used in a certain context.

7.3.1.3 Results publicly available

An evaluation of an OSS product is typically performed by a product’s end-user or a product integrator. However, it is useful if results from a previously performed evaluation (by other evaluators) are also available to the user. Over time, the number of evaluation results can grow to become a significant body of knowledge about a particular OSS product, which is a result of independently performed evaluations by different users. The availability of this collection can save a user from the effort to perform the evaluation him- or herself. Furthermore, if the user prefers to evaluate the OSS product independently, the available results can be crosschecked. This may give the user an increased confidence in the evaluation result (if the results are similar), or give the user a hint to look into the product in more detail (if the results are contradictory).

7.3.1.4 Method’s relation to other methods

The proposal of an OSS evaluation method is often the result of efforts to improve or complement some existing methods or approaches. It is usually important to understand the relationship between different methods. Such an understanding can enable practitioners to focus on identifying the similarities and differences between related methods and select a method that is most suitable to their needs. The relation between methods is not always made explicit; the design of a method may be influenced by the reported related work. Therefore, two types of
relationships are considered: “mentions”, meaning that a paper presenting a method mentions (and possibly discusses the limitations of) a method as related work, and “based on”, meaning that the method is based on the listed methods. This information was extracted as reported by authors presenting the methods.

7.3.2 Method User

The second component of FOCOSEM is “method user”, and contains comparison elements regarding the intended method’s user. The comparison for this component is based on two elements: a user’s required skills and a method’s intended users. These are discussed next.

7.3.2.1 Required skills

Different methods may involve different levels of complexity. Therefore, the required skills needed for using different evaluation methods may also differ. If the evaluation method requires too many skills, the method may not be suitable in a particular context. Upfront knowledge of a method’s required skills is therefore useful while choosing an OSS evaluation method.

7.3.2.2 Intended users

An OSS evaluation method is to be used by one or more users or stakeholders. Before choosing a particular method, it is useful to understand what stakeholders have a role in that method’s defined process. If a method requires involvement of different people, the method may not be suitable if limited resources are available for the OSS evaluation. Therefore, knowing upfront the kinds of stakeholders to be involved can help the selection of a suitable OSS evaluation method.

7.3.3 Method Process

The third component of the framework is concerned with the method’s process, which can be mapped to NIMSAD’s “problem-solving” process. This component contains elements to compare the way in which methods
approach the OSS evaluation. The elements are discussed in more detail next.

7.3.3.1 Activities

All methods define a sequence of activities, or steps, usually organised in a number of evaluation phases. The number, nature and granularity of these steps may vary significantly according to a given approach. Insight into what activities are involved may help the practitioner to assess the suitability of the method in a certain context.

7.3.3.2 Evaluation categories

The number of categories of evaluation criteria that is used to evaluate an OSS product indicates something about how many aspects of the product under evaluation are considered. The granularity of categories varies according to the evaluation method and also depends on the number of hierarchy levels of the evaluation criteria.

7.3.3.3 Number of criteria

The number of criteria that is used to evaluate a certain OSS product may give an indication about the rigour of the evaluation. For example, an evaluation based on twenty criteria can provide a more complete assessment than an evaluation based on five criteria. However, the researcher does not claim that the number of criteria is necessarily related to the quality or usefulness of an evaluation method. Some evaluation methods may organise the criteria in a tree-like fashion. In such cases, only the number of “leaf” criteria were counted.

7.3.3.4 Output

The output of an OSS evaluation method is usually the result whether or not to select a certain OSS. It is important to understand what the result of each method looks like as that can help practitioner to interpret the result. A result of an OSS evaluation can be as simple as a single value (for instance, a weighted average, such as delivered by the Multi-Attribute Utility Technique presented in (Wallnau et al., 2002, p.115)); or a result can also be more complex, for instance, a vector with several values. A
single value may not convey sufficient information and justification on which a selection decision for an OSS can be made.

7.3.3.5 Tool support
Some of the proposed evaluation approaches are supported by appropriate tools, which are expected to speed up the evaluation process. Tools may also allow the storage of evaluation results in a repository for later reuse. Tools can be as simple as a spreadsheet or as complex as a web-based evaluation system for supporting geographically distributed stakeholders. Evaluation sheet templates (such as created in a text processor) are not considered as tools; the focus is specifically on the availability of a software system that supports end-users in the evaluation process.

7.3.4 Method Evaluation
The last component of FOCOSEM is “method evaluation”. Jayaratna argues that evaluation is the most important element of the NIMSAD framework (Jayaratna, 1994, p.108). The evaluation component of FOCOSEM contains two elements: validation and maturity stage. These are discussed in more detail below.

7.3.4.1 Validation
Merely proposing a method, framework or approach does not suffice; it must be validated to increase practitioners’ confidence in a solution’s correctness, adequacy or generality (Shaw, 2001). The researcher defines three stages of validation:

1. No validation
2. Limited validation (e.g., through an example case study presented in a paper),
3. Extensive validation (method has been validated in several real world situations).

The level of validation may affect the confidence that method users may have in that method’s ability to satisfy its goals of evaluation.

7.3.4.2 Maturity stage
The first efforts to systematically address OSS evaluation were made in 2003, through the introduction of the first Open Source Maturity Model
(OSMM) by DuijnHouwer and Widdows (2003). In order to assess the maturity of an OSS evaluation method, the researcher has defined four maturity phases, which are based on the research reported by Ali Babar et al. (2004) and Kitchenham (2002). The work by Ali Babar et al. assumes that evaluation results appear in the literature, which is not applicable in the case of OSS evaluation methods, since OSS evaluation results are not published. Therefore, the researcher found it appropriate to adapt that classification to the following four stages, which are more appropriate in this research’s context:

1. Inception (method has recently surfaced and is not or little used);
2. Active (method is actively being used);
3. Inactive (method is still supported but not actively used);
4. Dormant (method is no longer supported or used).

### 7.4 COMPARISON OF OSS EVALUATION METHODS

This section presents the results of the comparison of six OSS evaluation methods, using the FOCOSEM framework.

#### 7.4.1 Method context

Table 7-3 presents the comparison of the OSS evaluation methods for the “context” component. Although the overall goal of all methods is to assess the quality of an OSS product, each method has a specific goal. The researcher extracted this information for each method from the papers as referenced in Table 7-1.

Table 7-3 shows that Capgemini and Navica OSMMs both focus on “maturity”, but Capgemini’s model also takes into account business requirements, whereas Navica’s model promises a “quick” assessment. QSOS was designed for evaluating OSS in an objective, traceable and argued manner. OITOS mentions to limit and control risks involved in adoption of OSS. OpenBQR is explicitly focused on evaluating aspects “relevant to the user”, whereas SQO-OSS’s goal is to do the evaluation automatically with as little human intervention as possible.
Three of the methods (Navica OSMM, QSOS and OpenBQR) include functionality as a criterion for the evaluation of OSS. The other methods thus assume that the OSS being assessed conforms to the functional requirements.

OpenBQR and SQO-OSS do not provide a public repository of OSS evaluation results. The Navica OSMM website provides only a few evaluation reports. The websites of Capgemini OSMM, OITOS and QSOS on the other hand provide the results from numerous previous evaluations.

The OSMMs by both Capgemini and Navica are not based on any other previous work. They were the first evaluation methods proposed in 2003 and 2004 respectively. The QSOS method does not mention any related work either. OpenBQR was defined as an extension and integration of OpenBRR and QSOS (Taibi et al., 2007). The OITOS method defines an evaluation model that is based on Navica OSMM, Capgemini OSMM and OpenBRR. The work presenting SQO-OSS only mentions, but does not report to be based on OpenBRR, QSOS and Navica OSMM.
Table 7-3. Comparison for the “Method context” component of FOCOSEM.

<table>
<thead>
<tr>
<th>Method</th>
<th>Specific goal</th>
<th>Functionality evaluation</th>
<th>Results publicly available</th>
<th>Relation to other methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capgemini Open Source Maturity Model (OSMM)</td>
<td>To determine an OSS product’s maturity and match to business requirements, compare a product with commercial alternatives and show the importance of an Open Source Partner</td>
<td>No</td>
<td>Yes (extensive)</td>
<td>n/a</td>
</tr>
<tr>
<td>Navica Open Source Maturity Model (OSMM)</td>
<td>To make a quick assessment of the maturity level of a given open source product.</td>
<td>Yes</td>
<td>Yes (limited)</td>
<td>n/a</td>
</tr>
<tr>
<td>QSOS</td>
<td>To qualify, select and compare free and open source software in an objective, traceable and argued way.</td>
<td>Yes</td>
<td>Yes (extensive)</td>
<td>n/a</td>
</tr>
<tr>
<td>OITOS</td>
<td>The project aims at strategic evaluations of Open Source solutions, in order to limit and control the risks for enterprise IT adoption.</td>
<td>No</td>
<td>Yes (extensive)</td>
<td>Based on: Navica OSMM, Capgemini OSMM, OpenBRR</td>
</tr>
<tr>
<td>OpenBQR</td>
<td>To support the potential user in the evaluation and choice of OSS in a flexible way, taking into account all the aspects that are relevant to the user.</td>
<td>Yes</td>
<td>No</td>
<td>Mentions: Navica OSMM; based on: OpenBRR, QSOS</td>
</tr>
<tr>
<td>SQO-OSS</td>
<td>To support an automated software evaluation system, with minimal human intervention.</td>
<td>No</td>
<td>No</td>
<td>Mentions: OpenBRR, QSOS, Navica OSMM</td>
</tr>
</tbody>
</table>
7.4.2 Method user

Table 7-4 shows the comparison for the “user” component of FOCOSEM, based on two comparison elements: “required skills” and “intended users”.

None of the methods explicitly mentions the skills required to use the methods. As was argued in Sub-section 7.3.2.1, it is important for a potential method user to understand the level of expertise and knowledge required for using a particular method before choosing that method.

The researcher observed that sometimes the intended user was not made explicit. In such situation, the researcher tried to surmise this information from the content of the paper reporting the method. All methods are intended to be used by IT organisations that need to select OSS products. QSOS also explicitly mentions OSS developers (communities), as does OpenBQR. Capgemini OSMM is the only method intended to be used in close collaboration with a (Capgemini) consultant, together with end-users who need to select an OSS.

Table 7-4. Comparison for the “method user” component of FOCOSEM.

<table>
<thead>
<tr>
<th>Method</th>
<th>Required skill</th>
<th>Intended users</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capgemini Open Source Maturity Model (OSMM)</td>
<td>Not specified</td>
<td>Capgemini consultant and end-user/customer</td>
</tr>
<tr>
<td>Navica Open Source Maturity Model (OSMM)</td>
<td>Not specified</td>
<td>IT organisations</td>
</tr>
<tr>
<td>QSOS</td>
<td>Not specified</td>
<td>Professionals, non-professionals, communities, IT-experts</td>
</tr>
<tr>
<td>OITOS</td>
<td>Not specified</td>
<td>Small and medium sized enterprises</td>
</tr>
<tr>
<td>OpenBQR</td>
<td>Not specified</td>
<td>Any of: ICT experts, OSS developers, software quality assurance and measurement professionals</td>
</tr>
<tr>
<td>SQO-OSS</td>
<td>Not specified</td>
<td>IT managers</td>
</tr>
</tbody>
</table>
7.4.3 Method process

Each method describes its activities at different levels of granularity and complexity. Some methods define sub-steps of the main activities. The researcher aims to provide an overview of various methods, and has therefore left those steps out in the results. For details, interested readers are referred to the methods’ sources in Table 7-1.

Capgemini’s OSMM by far has the most extensive and detailed set of activities. QSOS defines an iterative process consisting of four steps. SQO-OSS was designed for minimal human intervention, and thus has a strong focus on automated evaluation. The activities of that method primarily define the evaluation and aggregation models. The actual evaluation is performed automatically. The various methods have different sets of evaluation categories. Certain categories are common among the various methods, such as “product” and “support”. However, similar evaluation criteria may be classified in different categories, depending on the method’s classification. Interested readers are referred to the methods’ sources in Table 7-1 for details.

The number of criteria used by different OSS evaluation methods varies from 12 to 51. One could argue that a higher number of criteria reflects the quality of the OSS project more precisely. In that sense, Capgemini OSMM and OITOS have a less precise quality evaluation than OpenBQR and Navica OSMM, whereas SQO-OSS and QSOS are most extensive. SQO-OSS considers 10 attributes (e.g. “stability” and “testability”), and uses a total of 34 different metrics to quantify them. Capgemini uses 12 criteria, and has an additional list of 15 “application indicators”, which consider the suitability of the product for the environment in which it will be used.

OpenBQR, Navica OSMM and QSOS calculate the output as a weighted average sum to aggregate the scores of the criteria. SQO-OSS uses an ordinal scale for each of the 10 attributes used for evaluation. Capgemini OSMM does not specify how to calculate the final output.
score of the evaluation process. The OITOS method yields a vector of values for each evaluation category.

The methods QSOS, OpenBQR and SQO-OSS provide tool support. However, no further information or links are provided for the tool supporting OpenBQR presented in (Taibi et al., 2007). The tool for SQO-OSS (“Alitheia”) is the core of the method, since it is focused on automatic evaluation with minimal human intervention and an evaluation cannot be performed without tool support. Though the SQO-OSS project has finished, the tool was published as open source software and now has its own website at (Alitheia, 2006). QSOS is supported by a web-based tool called “O3S” (Open Source Selection Software, 2006).
Table 7-5. Comparison of the “Method process” component of FOCOSEM.

<table>
<thead>
<tr>
<th>Method</th>
<th>Activities</th>
<th>Evaluation categories</th>
<th>Number of criteria</th>
<th>Output</th>
<th>Tool support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capgemini Open Source Maturity Model (OSMM)</td>
<td>1. Product research and rough selection; 2. Scoring of products using product indicators; 3. Scoring using indicators by CG consultant; 4. Interview with customer on value of application indicators; 5. Scoring of application indicators together with consultant; 6. Determining scorecard per product and final selection 7. Evaluation</td>
<td>Product, use, integration, acceptance</td>
<td>12 criteria</td>
<td>Final score, not specified how to calculate.</td>
<td>No</td>
</tr>
<tr>
<td>Navica Open Source Maturity Model (OSMM)</td>
<td>1. Assess product elements’ maturity and assign a maturity score; 2. Define weights for each element based on organisation’s requirements; 3. Calculate overall maturity score.</td>
<td>Product software, support, product integrations, documentation, training, professional services</td>
<td>21</td>
<td>Weighted score</td>
<td>No</td>
</tr>
<tr>
<td>QSOS</td>
<td>Iterative: 1. Definition; 2. Evaluation; 3. Qualification; 4. Selection.</td>
<td>Intrinsic durability, industrialized solution, integration, technical adaptability, strategy</td>
<td>51</td>
<td>Weighted score</td>
<td>Yes</td>
</tr>
<tr>
<td>OITOS</td>
<td>1. Context analysis; 2. Preliminary selection; 3. Filtered selection.</td>
<td>Development, community, transition, technology</td>
<td>13</td>
<td>Vector of values for each category</td>
<td>No</td>
</tr>
<tr>
<td>OpenBQR</td>
<td>1. Quick assessment filter; 2. Data collection &amp; processing; 3. Data translation</td>
<td>Target usage, external qualities, internal qualities, support, cost, functionality</td>
<td>17</td>
<td>Weighted score</td>
<td>Yes</td>
</tr>
<tr>
<td>SQO-OSS</td>
<td>1. Definition of evaluation model; 2. Definition of aggregation method</td>
<td>Product quality, community quality</td>
<td>34 metrics</td>
<td>Ordinal scale (good, fair poor), for each attribute</td>
<td>Yes</td>
</tr>
</tbody>
</table>
7.4.4 Method evaluation

The comparison for the elements of the “evaluation” component is shown in Table 7-6. Three of the compared methods (Navica’s OSMM, OpenBQR and SQO-OSS) can be considered as limited validated. The other three methods (Capgemini OSMM, QSOS and OITOS) have been extensively validated. It has already been argued that the validation of a method is important as it increases the method user’s confidence to use the method.

Table 7-6. Comparison of the “Evaluation” component of FOCOSEM.

<table>
<thead>
<tr>
<th>Method</th>
<th>Validation</th>
<th>Maturity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capgemini Open Source Maturity Model (OSMM)</td>
<td>Extensively</td>
<td>Inactive</td>
</tr>
<tr>
<td>Navica Open Source Maturity Model (OSMM)</td>
<td>Limited</td>
<td>Inactive</td>
</tr>
<tr>
<td>QSOS</td>
<td>Extensively</td>
<td>Active</td>
</tr>
<tr>
<td>OITOS</td>
<td>Extensively</td>
<td>Dormant</td>
</tr>
<tr>
<td>OpenBQR</td>
<td>Limited</td>
<td>Dormant</td>
</tr>
<tr>
<td>SQO-OSS</td>
<td>Limited</td>
<td>Dormant</td>
</tr>
</tbody>
</table>

In order to identify the maturity stages, the researcher browsed through different methods’ websites. If the researcher was not able to find recent information about a method, the method was classified as “dormant”. The researcher could not identify recent information or updates for OITOS, OpenBQR and SQO-OSS. The OSMMs of Capgemini and Navica still seem to be actively supported, but none of these methods seems to be actively used, which is why these were classified as “inactive”. Capgemini’s OSMM does not have publicly available evaluation results, and is used by Capgemini itself. Hence, this method may be actively used, however, the researcher could not find any evidence of its use. The QSOS method seems to be the only one that is actively used, with new OSS evaluations being reported on the website.
7.5 CHAPTER SUMMARY

A variety of OSS evaluation frameworks and methods have been proposed that can provide guidance to practitioners to select suitable OSS products. Research has shown that these methods are rarely used. Such a lack of adoption could be caused by a lack of awareness among practitioners: they may not know that such methods exist. Another cause could be a deficiency in clarity of the various methods: practitioners may just not be able to select an appropriate method. In order to gain an understanding of how these methods differ and relate to one another, this chapter provided a systematic comparison based on a framework, named FOCOSEM. The framework consists of four components, each of which has a number of elements that address certain aspects of the evaluation method. The framework was used to perform a comparison of six OSS evaluation methods, the results of which are presented in this chapter.

The results of the comparison, in particular in the category “Method Process” (see Sub-section 7.4.3) also indicated that existing methods consider a wide array of an OSS product’s attributes (such as its developing community), and some methods also consider a product’s quality attributes such as performance and scalability, or “ilities”. As Chapter 4 has outlined, quality attributes are directly affected by a product’s software architecture. None of the methods that were examined in the framework-based comparison, however, consider the product’s architecture. Some methods do consider “architectural” aspects, such as the availability of third-party plug-ins, but such information does not help in a quality assessment. Though quality attributes can, of course, be measured (e.g., a product’s reliability may be measured by its Mean Time To Failure (MTTF)), insight into a product’s architecture would be quite helpful to assess its architectural fit within a larger system.

Whether or not practitioners are interested in architectural knowledge of a product is a topic that has not been explored. It may well be that practitioners do not have an interest in such architectural knowledge, and that existing evaluation methods would suffice. In order
to investigate the need for architectural knowledge of OSS products, Chapter 8 which follows presents the results of an explorative, qualitative survey.
Chapter 8

The Importance of Architectural Knowledge for Open Source Software Integrators*

8.1 INTRODUCTION AND CHAPTER LAYOUT

Chapter 7 presented an analysis of OSS evaluation methods that have been proposed to assist practitioners in the task of selecting suitable OSS products. Existing methods consider a wide array of an OSS product’s attributes (including characteristics of its developing community), and some methods also consider a product’s internal quality, or “ilities”. Such quality attributes (QAs, e.g., performance, security) can provide important insights to practitioners that consider the adoption of an OSS product. Though some of the evaluation methods consider these QAs, none of them consider the architectural knowledge (AK) of the products (e.g., patterns). As outlined in Chapter 4, a software product’s architecture has a direct effect on the product’s QAs. However, as mentioned, existing OSS evaluation methods do not consider knowledge of a product’s architecture. Much of the literature on OSS selection suggests that a products’ longevity and potential for future support are important factors in the selection of OSS (e.g., (Ven and Verelst, 2009)). Practitioners may not consider AK of an OSS product to be valuable. To investigate this issue, this chapter addresses the following research question:

What is the importance of architectural knowledge for OSS integrators?

In order to address this question, the researcher conducted a small-scale survey. This chapter presents the detailed design and results of this study. The chapter is structured as follows. Section 8.2 presents the design of the study conducted to address the above research question. Section 8.3 presents the results of the study; this section is divided into four sub-sections, each of which addressed one of the four specific research questions. A discussion of the results is presented in Section 8.4. Limitations of the study are discussed in Section 8.5. The chapter is concluded in Section 8.6, which summarises key implications of this study.

8.2 STUDY DESIGN

This section presents the design of this study. Sub-section 8.2.1 presents the four specific research questions. The data collection method for this study is presented in Sub-section 8.2.2, followed by a discussion of the data analysis procedure in Sub-section 8.2.3.

8.2.1 Research questions

As mentioned above, this study was designed to address the general research question:

_What is the importance of architectural knowledge for OSS integrators?_

Using Punch’s framework of research questions presented in Chapter 5, this question can be considered to be a general research question, and as such is too high-level to answer directly. In order to answer this research question, a number of specific research questions (SRQ) are derived, which can be directly answered. This section presents these specific research questions.

Much of the literature on OSS selection highlights the importance of evaluation criteria such as the OSS license, support, functionality, and so on. However, there are few studies from the perspective of OSS integrators. There is no empirical evidence about the AK needs of OSS integrators. Since software architecture plays an important role in
component integration (see Chapter 4), it is important to investigate what architectural knowledge (AK) OSS integrators need. Therefore, the first specific research question in this study is:

**SRQ1: What AK of OSS products do integrators need?**

Besides an understanding of what AK is needed, it is also useful to understand how AK can help OSS integrators. Hence, the second specific research question is:

**SRQ2: Why do integrators need AK of OSS products?**

Furthermore, it is also important to find out whether the required AK about OSS component is usually available. Hence, the third specific research question is:

**SRQ3: Is AK of OSS products generally available to integrators?**

Throughout this thesis, the researcher asserts that having AK of an OSS product is important. However, it is equally important to find out OSS integrators’ perceived importance of AK in integration. In order to get an empirically based understanding of the importance of the role of architecture in OSS integration, the researcher defined the fourth specific research question:

**SRQ4: What is the relative importance of AK of OSS components?**

These four questions are respectively answered in Sub-sections 8.3.1 to 8.3.4.

### 8.2.2 Data collection

As outlined in Chapter 5, the researcher chose survey methodology for this study’s design. The reader is referred to that chapter for details of the discussion of the research methodology. This section focuses on the selection of the participants and briefly summarises the data collection and analysis procedures.
IT practitioners in the professional network of the researcher and his supervisors were invited to participate in the study; the sample was therefore a convenience sample (Robson, 2002). Table 8-1 lists background information of the participants. The participants are numbered P1 to P12; these identifiers are used throughout the remainder of this chapter.

All but one participant worked at organisations located in different countries in Europe; P5 was located in the USA. Specific details are not reported for confidentiality reasons. Two participants (P3 and P4) worked both at organisation C, but at different branches or departments. This is indicated by a number suffix (i.e., C1, C2). The participants at organisation E were contacted through a professional contact at this organisation, who was also one of the participants. The participants had various positions, and all had extensive experience in the field. In total, data was drawn from 12 participants who worked at five different organisations. Though participants P6-P12 all worked at organisation E, they worked in different teams or departments (E1 to E6).

Table 8-1. Participants of the survey study.

<table>
<thead>
<tr>
<th>ID</th>
<th>Position</th>
<th>Experience</th>
<th>Org.</th>
<th>Domain</th>
<th>Interview</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>Software architect</td>
<td>13 years</td>
<td>A</td>
<td>Business</td>
<td>Phone</td>
</tr>
<tr>
<td>P2</td>
<td>Software developer</td>
<td>5 years</td>
<td>B</td>
<td>Embedded</td>
<td>Face-to-face</td>
</tr>
<tr>
<td>P3</td>
<td>R&amp;D developer</td>
<td>13 years</td>
<td>C1</td>
<td>Telecom</td>
<td>Phone</td>
</tr>
<tr>
<td>P4</td>
<td>R&amp;D developer</td>
<td>13 years</td>
<td>C2</td>
<td>Face-to-face</td>
<td></td>
</tr>
<tr>
<td>P5</td>
<td>Independent consultant</td>
<td>10 years</td>
<td>D</td>
<td>Business</td>
<td>Instant messenger</td>
</tr>
<tr>
<td>P6</td>
<td>Software architect</td>
<td>20 years</td>
<td>E1</td>
<td>Safety critical systems</td>
<td>Face-to-face</td>
</tr>
<tr>
<td>P7</td>
<td>Project leader</td>
<td>26 years</td>
<td>E2</td>
<td>Safety critical systems</td>
<td>Face-to-face</td>
</tr>
<tr>
<td>P8</td>
<td>Sr software designer</td>
<td>10 years</td>
<td>E3</td>
<td>Face-to-face</td>
<td></td>
</tr>
<tr>
<td>P9</td>
<td>Team leader*</td>
<td>25 years</td>
<td>E4</td>
<td>Face-to-face</td>
<td></td>
</tr>
<tr>
<td>P10</td>
<td>Software architect</td>
<td>12 years</td>
<td>E5</td>
<td>Face-to-face</td>
<td></td>
</tr>
<tr>
<td>P11</td>
<td>Technology manager</td>
<td>25 years</td>
<td>E2</td>
<td>Face-to-face</td>
<td></td>
</tr>
<tr>
<td>P12</td>
<td>Team leader*</td>
<td>15 years</td>
<td>E6</td>
<td>Face-to-face</td>
<td></td>
</tr>
</tbody>
</table>

* Participants have also experience as a software architect.
The participants worked at organisations in different domains that integrate OSS products in various degrees. Organisation A is a large consultancy organisation that develops business process support systems. Organisation B develops mostly embedded software. Organisation C is a research and innovation institution that develops both proof-of-concept prototypes and final products for customers. Participant P5 (organisation D) is an independent consultant, and works on business process support systems. Organisation E develops hardware and software for safety critical systems.

Prior to conducting the interviews, the researcher designed an interview guide (Taylor and Bogdan, 1984), which is included in this thesis as Appendix A. The researcher conducted all face-to-face interviews at the premises of the participants’ organisations. All but one interview lasted approximately one hour; the interview with P5 (through instant messenger (IM)) lasted approximately two hours. All face-to-face and phone interviews were digitally recorded with the participants’ consent. The interviews were transcribed verbatim by the researcher (the transcript of the interview with P5 was automatically recorded by the IM client), resulting in more than 120 pages of text (A4 size).

8.2.3 Data analysis

As outlined in Chapter 5, the data were analysed using qualitative data analysis methods (Seaman, 1999). The researcher thoroughly read all interview transcripts, during which phrases of interest were extracted that were relevant to one of the four research questions. The extracted data was stored in a spreadsheet document, along with the page number of the source transcript that allowed the researcher to trace back phrases to their original context. The researcher annotated each entry with a code, or label, that reflected the contents of the entry. After this, the data were sorted and grouped based on the codes. Per group, the researcher used the constant comparison technique (Seaman, 1999) to identify common themes for answering the research questions.
8.3 RESULTS OF THE SURVEY

This section presents the results of the study. Research questions SRQ1 to SRQ4 defined in Sub-section 8.2.1 are addressed in Sub-sections 8.3.1 to 8.3.4 respectively.

8.3.1 Architectural Knowledge Needs of OSS Integrators

The researcher identified four categories of information that integrators would like to have. Table 8-2 presents an overview of the findings using the four categories (numbered N1 to N4) emerged from the coding of the participants’ answers. It is interesting to note that the first type of AK (N1) affects the other three types of AK (N2-N4); for instance, a component’s structure directly affects its quality attributes and behaviour. This is discussed in more detail in Section 8.4. The remainder of this subsection discusses the four categories (N1-N4) in more detail.

Table 8-2. Types of architectural knowledge needed by OSS integrators.

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
<th>Reported by</th>
</tr>
</thead>
<tbody>
<tr>
<td>N1</td>
<td>Component structure. Patterns, partitioning, structure</td>
<td>P1, P2, P3, P4, P5, P6, P7, P10, P12</td>
</tr>
<tr>
<td>N2</td>
<td>Quality attributes and behavior. Information about performance, e.g., bottlenecks, processor usage, disk usage, and other resources; reliability, stability, robustness, predictability, use of tactics.</td>
<td>P2, P4, P5, P6, P7, P8, P10, P11</td>
</tr>
<tr>
<td>N3</td>
<td>Architectural fit. Ease of integration, interface, API compliance, dependencies</td>
<td>P2, P3, P4, P5, P6, P7, P8, P10, P11</td>
</tr>
<tr>
<td>N4</td>
<td>Component usage. Insight in how the component can be used and how it performs the task</td>
<td>P2, P4, P10</td>
</tr>
</tbody>
</table>

8.3.1.1 Component structure

Most participants were interested in the internal structure of components. When speaking of a component’s internal structure, participants often spoke of ‘pattern’. In particular, some participants mentioned the presence of patterns such as layers and model-view-controller to be interesting for their purpose. One participant stated:
“Well, the first thing that I would do is to look at the documentation, in which they explain the structure there, preferably in pictures. And when those are not available, then it’s a matter of extracting a zip file [jar file] and see how the directory structure would look like.” —P4, R&D developer.

8.3.1.2 Quality attributes and behaviour
Several participants explicitly indicated specific QAs, such as performance, reliability and stability. Like the literature on QAs, the participants also agreed that these QAs are directly affected by a component’s behaviour related to the use of resources (e.g., memory, processor, interrupts, database connections). Integrators consider it to be important to understand how a component behaves regarding these system resources; one participant reported to have used a “sandbox” to measure performance, processor usage, disk usage, and similar parameters.

One factor that affects a component’s behaviour is the use of architectural tactics (Bass et al., 2003; Harrison and Avgeriou, 2010). A tactic is a common technique to achieve a certain quality attribute. For instance, in order to improve the performance of software that connects to a database system, a developer may apply the “connection pooling” tactic. Such information is valuable to integrators, as one participant described:

“If it is a component that uses relatively expensive or limited system resources, then you’d like to know the strategy of those components to deal with resources. If I'm integrating it and it's using 10 out of 11 available database connections […] that's certainly handy to know.” —P10, Software architect.

The researcher did not quantitatively analyse the most important quality attributes such as in (Ameller and Franch, 2010), as the sample size of this study was limited, and the relative importance of QAs is likely to be dependent on the domain in which the software operates.

8.3.1.3 Architectural fit
The most common concern of participants was the architectural fit of a component; in other words, does a component fit in the existing architecture? One participant described it clearly:
“[I look at] whether it can be used in the architecture that I had envisioned. To what extent do I need to adapt my own software in order to be able to use that product? [And] if I have to make certain changes in an OSS product, in order to be able to use it, how much.” —P3, R&D developer.

Participants used the term architectural fit, but also referred to the interface and API of a component. It was interesting to find that a large majority of the participants distinguished a component’s architectural fit from its internal structure or used patterns (see Sub-section 8.3.1.1 above), and generally considered the former to be more important than the latter. However, it is well known that patterns have a direct influence on a component’s architectural fit (Shaw, 1995a).

8.3.1.4 How to use a component
A few participants mentioned the need to understand how they can use a component. Besides insight in whether the component fits (see Sub-section 8.3.1.3), practitioners need to understand how it can be used within the system they build, and how to access the functionality provided. The participants prefer to have examples of how to use the software, or even the availability of a test environment that demonstrates how a component can be used, as one participant suggested:

“If the component came together with a sort of test environment, a sort of test application, that could show how the components can be used, and how it performs the tasks that it is supposed to do. That would help a lot.” —P2, software developer.

8.3.2 Why is Architectural Knowledge Needed?
The previous section addressed the question what AK integrators need. This section addresses the question why integrators need AK. The researcher identified a number of different reasons how such knowledge can help integrators. Table 8-3 presents an overview of the findings to answer this question; these reasons (numbered W1 to W4) are discussed in more detail in the remainder of this sub-section.
Table 8-3. Reasons why architecture knowledge is important to integrators.

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
<th>Reported by</th>
</tr>
</thead>
<tbody>
<tr>
<td>W1</td>
<td>Quality assessment. Architecture, partitioning and patterns indicate certain maturity and qualities and help to analyse a component from a performance and functional perspective.</td>
<td>P2, P5, P7, P8, P10</td>
</tr>
<tr>
<td>W2</td>
<td>Assess architectural fit. Internal structure and patterns indicate what architecture is used and whether it fits in the existing architecture</td>
<td>P1, P2, P3, P6, P7, P10</td>
</tr>
<tr>
<td>W3</td>
<td>Improve maintainability. Patterns improve maintainability and replacing of components</td>
<td>P1, P2, P7, P8, P10, P12</td>
</tr>
<tr>
<td>W4</td>
<td>Help to use component. Architectural information and patterns help you to use the component in a useful way</td>
<td>P1, P10</td>
</tr>
</tbody>
</table>

8.3.2.1 Quality assessment

An important use of AK is that it can be used in the assessment of a component with respect to its quality attributes. Participants particularly referred to the structure and patterns used in this context. Patterns are common solutions with predictable effects on a component’s QAs. This could refer to a product’s runtime attributes, such as performance and reliability, but also with respect to its build-time properties, such as the ease of integration. One participant explained:

“[Information of internal structure is] certainly very useful information, because such a pattern indicates what kind of architecture was used, and such a pattern indicates a certain quality. That could be configurability, but also the ease of integrating, those patterns secure a certain stability.” —P7, project leader.

8.3.2.2 Architectural fit assessment

Several participants indicated that architectural information provides useful insights into whether or not an OSS product is compatible at the architectural level with the main system in which it may be integrated. As noted by one participant:

“Well at least it indicates what style is being used, and what are the fundamental concepts, and your own product, or platform in our case, also has certain styles, and then you can see, whether it fits together. Does it make sense to try to put it together, or will we need a whole lot of glue code to be able to use it. And if so, is it worth our while, or is it easier to develop ourselves. At a high level it indicates whether something is a good match with your product, with the existing software.” —P10, software architect.
8.3.2.3 Maintenance and evolution

A number of participants indicated that knowledge of the patterns used in a component could help in the maintenance phase of a product, including defect fixing. Patterns increase the understandability of how a component is constructed, which makes it easier to change the component. One participant described this as follows:

“If you have some OSS that's constructed using good patterns, then of course it'll help you to make changes. If you get OSS that's constructed in an ad-hoc manner, organically grown, and you decide to make changes to make it fit better, then that's completely useless. The more it is structured, thought about it in terms of layers and standard structures, the easier it will be to adopt it and change it.” —P12, team leader.

While the actual presence of patterns improves a product’s maintainability, it is important to understand which patterns have been used, in order to make changes that do not degrade the pattern’s integrity and consistency.

8.3.2.4 Using the component

A good knowledge of architectural aspects of a component can help an integrator to understand the appropriate use of that component. During the study, a few participants also indicated that AK such as the patterns used provides valuable insights into whether or not a component will be suitable in the context of a given system and how to use that component. One of the participants explained:

“If you wouldn't know those [architectural concepts and patterns], it's difficult to use such a framework in a useful way. So in that case it would be difficult to find everything in the code, so at least you'd like to see some high level descriptions of the concepts and design patterns.” —P10, software architect.

8.3.3 The Availability of Architectural Knowledge

This section addresses the question whether or not AK is usually available to OSS integrators. The researcher identified two themes, namely (1) the general availability of AK and (2) how integrators deal with the lack of AK. The researcher decided to present this analysis in a
descriptive way, rather than a tabular representation used in Sub-sections 8.3.1 and 8.3.2.

8.3.3.1 **Availability differs per product**

The results indicate that whether or not AK is available, depends on the product. One participant highlighted that:

“[for] open source it's quite easy to figure out how it fits into my architecture. If I look at Spring, how does this fit into my architecture, is it MVC or part of MVC… I also think that is indirectly a big plus for OSS, because they actually need to be compatible, they need to be able to integrate with other parts at the architecture level, otherwise they wouldn't survive.” —P1, software architect.

This suggests that AK is sufficiently available. Another participant confirmed that for well-known frameworks such as the Spring framework, this information is indeed quite easily available, however, this may not be the case with other components:

“In case of Spring yes, but in other components not so much, and you just have to look in the documentation and in some cases even in the implementation.” —P3, R&D developer.

Participants P5 and P7 explicitly claimed that there is typically not much AK of OSS products available. One participant emphasised that a good design will result in good quality code, and makes the author’s intents clear:

“I don't usually have much architectural info when investigating an OSS component, unless it's in the docs, on the website etc. Historically, there hasn't been a focus on architecture patterns by OSS component authors. It's not a best practice to include architectural patterns information.” —P5, independent consultant.

8.3.3.2 **How integrators deal with a lack of architectural knowledge**

The participants of the study mentioned different approaches to deal with a lack of AK of an OSS product. The main findings are presented under

---

24http://www.springsource.org/
the following categories: “Look at others”, “Assume the worst”, and “Try to recover”.

**Look at others.** One way to assess the quality of a product’s QAs, is to look at other customers of the component, as explained by one participant:

“Our line of thought with respect to the ‘ilities’ was, as long as a rather large group of people uses it, you may assume that most of the trouble with the ‘ilities’ are solved.” —P6, software architect.

**Assume the worst.** Another approach to deal with a component’s characteristics, such as security, is to assume the worst, and make sure that the rest of the system compensates for its shortcomings:

“I don't think I would try to understand whether a certain OSS product, whether it's secure enough. I think I would assume that it's not secure enough and then I would make sure that the environment in which the software is run takes care of the security.” —P3, R&D developer.

**Try to recover.** If architectural information in the documentation is missing, then the only solution to this is to look into the implementation or to ask the community for more information. However, in such a case, the time to get a reply was considered to be very important due to development schedules and deadlines. Furthermore, while studying the implementation was mentioned as a solution, it should be considered as a last resort:

“When you have to go into the code to know how it works, then I do think you have a problem. […] I think open source has much potential but the investment in knowledge is quite an issue.” —P9, team leader.

### 8.3.4 The Relative Importance of Architectural Knowledge

The last research question investigates the participants’ perceived importance of AK. This puts the need for AK in perspective compared to other selection criteria, which are known to be important, such as
availability of support and license. One participant considered AK as just one type of information that is needed, but stated that:

“[…] the more you know, the better.” —P10, Software architect.

As for SRQ3, the researcher identified a few themes, which are discussed next.

8.3.4.1 Missing AK hinders usage
In Sub-section 8.3.3.1 above, participant P1 indicated that the availability of AK of an OSS product is important for the project’s survival. Three participants explicitly indicated that the lack of an OSS component’s AK means that it drops on the list of candidates, and hinders its usage. This means that a lack of available AK negatively affects the selection decision of whether to use the OS product. One participant explained:

“Well, then you're making a big investment if you do that. So I can think it can hinder you in using it. When you have to go into the code to know how it works, then I do think you have a problem.” —P9, team leader.

8.3.4.2 Need for AK depends on product type and size
A recurring factor that influenced the participants’ interest in having AK of an OSS product is the type and size of the component. The information needs heavily depend on whether it is a library that provides functionality or non-critical parts, such as graphical widgets, or that it is a central component that makes up an important part of a system. One participant described it as follows:

“If I'm looking for an MVC framework, then yes [I'll be interested in architectural information]. But if I’m looking for a foo-munging module, and there are 12, then the last 3 release dates, smoke reports, browsing source, etc., are likely more important.” —P5, independent consultant.

A few participants explicitly highlighted the architectural impact that an OSS component may have on the existing system architecture. One participant described:
“But for instance the reporting engine, if we choose product A or B, that was more of a feature level, and not typically on the architecture level, even if we would pick product A or B, it wouldn't ruin our existing architecture because this is more on the side.” —P1, software architect.

Related to this issue is whether practitioners value a component’s functionality or architecture. On the one hand, some participants highlight the focus on functional compliance; one participant explained this concisely:

“My software MUST meet the functional requirements, and MAY have a good architecture.” —P5, independent consultant (emphasis by participant in IM transcript).

Furthermore, practitioners’ interest in a component’s functionality or architecture also depends on the type of the component. One participant explained:

“Well, frankly it's always been the functionality for us than that we're looking at the architecture. That may have to do with the type of components that we integrate. They're typically not complete subsystems, but rather limited libraries. Just those things that don't really count.” —P10, software architect.

On the other hand, some participants preferred a well-designed component with a good architecture to the functionality provided by the component. Extensibility and size of the component are decisive factors, as one participant explained:

“I would tend to choose for good quality and architecture, but perhaps a lack of functionality. But there should be a good possibility to extend that functionality, and when that's not there, then I may decide to take a chance and select the thing with all functionality but less quality, and to fix that thing myself. That also depends on the size by the way, when it's a huge project, then I won't start on that.” —P4, R&D developer.

8.3.4.3 The relative importance of architectural knowledge
So far, the answer to the question whether or not AK is an important factor is: “it depends”. Factors are the type and size of a component (e.g., a library that provides general functionality or graphical widgets versus a
framework that prescribes the structure of the overall software). In general, the participants of the study seem to be quite interested in a component’s AK, which can provide valuable insights that may affect the decision to use the component. One participant summarised this as follows:

“It depends on what it is, but I think if it is something that you're interested in anyway, in how it's constructed, which can be part of the evaluation of the piece of open source, then it's certainly interesting. It can give you a good feeling that people have thought about it. It fits or doesn't fit with what we have. So yes, it's certainly valuable.” —P10, software architect.

The participants generally agreed that the quality of a component can have many aspects, but that the critical selection factors depend on the context. One participant highlighted the importance of the implementation language in these words:

“The use of patterns in OSS components played some part when considering their use, but the flexibility of Perl allows integration of widely-varying programming models, so for Perl components, the test coverage, documentation and community support were more important.” —P5, independent consultant.

8.4 DISCUSSION OF RESULTS

In the previous section, four types of AK of OSS needed by OSS integrators were identified, as well as the main reasons for why AK is important. It is interesting to note that the categories identified in Section 8.3.1 (what AK is needed) and 8.3.2 (why AK is needed) are quite similar. Participants indicated the need of understanding a component’s structure, its quality attributes and behaviour regarding e.g., system resources, whether or not components are an architectural “fit” with the main system’s design, and how to use a component. Following are the main reasons why OSS integrators consider architectural knowledge valuable:

1. Assess the quality of the component;
2. Assess the architectural fit of the component;
3. Affect the maintainability of the component;
4. Understand how the component can be used.
Knowledge of a component’s structure (including its patterns) seems to be the most important aspect for integrators. While the participants indicated a desire to know a component’s QAs, its architectural fit (compatibility), and how to use it, it seems that the architectural structure of a component is valuable input to satisfy most AK needs. A component’s architecture structure, including its patterns, directly affects its QAs, architectural compatibility, and can provide insight on how to use the component.

The four categories of why an integrator would like to have AK of a component can be mapped to the three main phases in CBSD outlined in Chapter 4, namely Evaluation, Integration, and Maintenance. Quality assessment and assessment for architectural fit are both activities that are performed in the Evaluation phase, when components are evaluated and selected. Understanding of how to use a component is important in the Integration phase, when components are integrated into a product. Finally, having AK to improve a component’s maintainability supports the Maintenance phase of component integration, after the main system has been deployed. Therefore, these results suggest that AK can support the OSS integrator in all three phases of CBSD.

However, in Section 8.3.3 it was found that availability of AK of OSS products may vary, and just how much AK is needed also varies. When AK is not available, OSS integrators typically do not try to recover it, which means they have to take a different strategy to deal with such lack of information. From Section 8.3.4 it has become clear that practitioners do consider AK to be valuable, but the extent to which depends on the type and size of the product. A lack of AK, however, was shown to be a potential obstacle for using a component. These results highlight the importance to investigate how OSS integrators can be supported.
8.5 LIMITATIONS OF THIS STUDY

The researcher is aware of a few limitations of this study. Firstly, the study was based on data gathered through 12 interviews, which is insufficient to draw general conclusions. However, since the role of SA has not been studied extensively in the context of OSS integration, the researcher decided to perform an exploratory study. Once this field has matured and specific hypotheses have been defined based on initial findings, other types of studies with larger numbers of participants will be more appropriate, such as questionnaire-surveys.

Secondly, the sample of participants was a convenience sample, which means there is a selection bias. Participants were contacted through the professional network of both the researcher and his supervisors. Furthermore, seven participants worked at one organisation that is active in a safety critical domain; this may have biased the participants’ views towards certain concerns. However, the researcher did not find significant differences with respect to the participants’ opinions and needs of AK.

8.6 CHAPTER SUMMARY

The results of the survey study presented in this chapter addresses four specific research questions. The findings for each of these questions are briefly summarised in this section.

The first research question investigated what architectural knowledge OSS integrators need. Four types of AK were identified, namely (1) a component’s structure, (2) its quality attributes and behaviour, (3) insights in its architectural fit with the rest of the system, and (4) knowledge of how to use a component.

The second research question investigated reasons why integrators need AK of OSS components. Four main reasons emerged from the interviews. These are: (1) to perform quality assessment, (2) to assess architectural fit, (3) to support maintainability, and (4) to understand how a component can be used.
The third research question investigated whether AK is available for OSS components. It was found that the availability of AK for an OSS product depends on the product. In particular for large and well-known products, such knowledge is available, whereas for lesser-known products it may not be available. When AK is not available, practitioners deal with this in different ways. Three approaches were identified: (1) look at what other customers use; if a component has a sufficiently large group of users, then some practitioners assume the product’s quality is sufficient; (2) assume that a product’s quality is not sufficient, and take appropriate measures, for instance by making sure the rest of the system can compensate for the component’s shortcomings, and (3) try to recover architectural knowledge.

The fourth and last research question addressed by this study investigated the relative importance of AK in OSS integration. Three themes were identified. Firstly, missing AK of an OSS product means that the product may drop on the list of candidate products, and therefore may hinder its usage. Secondly, the need for AK depends on the product type and size. In particular, whether a product is a library that offers a set of functionality, or a central component that makes up an important part of the final system, is an important factor. Thirdly, participants generally agreed that a component’s quality has many aspects, but that the selection criteria are dependent on the context in which the component is to be used.

Concluding this chapter, while the importance of AK of a product depends on a variety of factors, the results of this study shows that, in general, practitioners are quite interested in AK of an OSS component. In particular insights into a component’s structure and patterns was considered to be useful information. However, as was discussed in Chapter 4, current techniques and approaches to identify architectural patterns have limitations. Chapter 9 which follows next presents the design and evaluation of a reference process to identify architectural patterns in OSS products. The researcher argues that identified architectural patterns can provide important hints regarding a product’s
quality attributes, which a practitioner can use in the evaluation of an OSS product.
Chapter 9

A Process for Identifying Architectural Patterns in Open Source Software*

9.1 INTRODUCTION AND CHAPTER LAYOUT

Chapter 4 discussed the importance of software architecture (SA) in component-based software development. In CBSD, an important task is the selection of appropriate components, which is, as Chapter 6 reported, one of the most reported challenges. Existing OSS evaluation methods (presented in Chapter 7) have limited attention for architectural knowledge (AK) of OSS products. One important type of AK is architectural patterns (see Chapter 4), as is confirmed by the findings presented in Chapter 8. Practitioners consider knowledge of architectural patterns to be quite useful in the assessment of OSS products. Patterns are also considered by Harrison and Avgeriou (2011) in their Pattern-Based Architectural Review (PBAR) method as an effective means to perform architectural reviews (see Chapter 4). While knowledge of architectural patterns is considered to be useful information, identifying architectural patterns is a challenging task. Existing pattern identification tools mostly focus on object-oriented design patterns. This chapter presents the design and evaluation of a reference process for identifying architectural patterns in OSS products. The process is a systematic approach to gather

information of an OSS product that can assist in the identification of architectural patterns.

The chapter is structured as follows. Section 9.2 presents an overview of the research design, which followed the Improvement Paradigm; this is an iterative research approach as discussed in Chapter 5. Section 9.3 presents the design of the first iteration, followed by Section 9.4, which presents the results of the first iteration. Based on the insights gained from the first iteration, the researcher designed a process for Identifying Architectural Patterns in OSS, named “IDAPO”. The design of the second iteration is presented in Section 9.5, followed by the results of in Section 9.6. In order to evaluate the effectiveness of IDAPO, the researcher conducted a quasi-experiment, which is reported in Section 9.7. Section 9.8 presents future work, and Section 9.9 concludes this chapter.

9.2 OVERVIEW OF THE RESEARCH DESIGN

Chapter 5 presented the six general research questions that this thesis addresses. This chapter address research question four, which is:

*RQ4: How can practitioners be guided in identifying architectural patterns in OSS products?*

The objective of this research is the development and evaluation of a reference process to guide practitioners in the task of identifying architectural patterns in OSS products; the resulting process is named “IDAPO”. The research design followed the “Improvement Paradigm” process, as was discussed in Chapter 5. The researcher performed three iterations of that Improvement Paradigm process; in each iteration the researcher conducted a study to (1) understand the situation, (2) assess what improvements can be made, and (3) to package the results of the study in order to integrate in the “environment”. The three iterations are shown in Figure 9-1.
Figure 9-1. Overview of the three iterations of the Improvement Paradigm.

The goal of the first iteration was to understand how architectural patterns are identified without any form of support. Based on identified approaches and challenges, the researcher assessed how practitioners could be guided in this task. Based on this, the researcher packaged the results by formalising these approaches in a reference process definition. These results are presented in Section 9.4. The goal of the second iteration was to further the researcher’s understanding of the process of
identifying architectural patterns in an OSS product. Through additional interviews, the researcher could cross-validate and assess how to enhance the initial version of IDAPO (v1) by triangulating across another set of data. These new insights were packaged into an enhanced version of IDAPO (v2), which is presented in Section 9.6. The third iteration aimed at understanding how well IDAPO (v2) can support the task of identifying architectural patterns. This was done by means of a quasi-experiment. The design and results of the third iteration is presented in Section 9.7.

The three studies were performed over a time period of approximately two years; the time schedule of the three studies is shown in the timeline in Figure 9-2.

Figure 9-2. Timeline of studies conducted for this research.

The researcher drew from three data sources for the study in the first iteration (February 2009 to September 2009), namely interviews, reflection reports, and a write-up of the researcher’s own personal experiences. The study for the second iteration was conducted in March 2010; data were collected through another set of interviews. The study for the third iteration (an evaluative experiment) was conducted in December 2010; data for this third iteration were collected by means of a quasi-experiment.

9.3 DESIGN OF ITERATION 1: EXPLORING APPROACHES, CHALLENGES AND SUGGESTIONS FOR IMPROVING PATTERN IDENTIFICATION

This sub-section presents the design of the study for the first iteration of the Improvement Paradigm. The study goal and specific research questions for this study are presented in Sub-section 9.3.1; the data
collection procedure is presented in Sub-section 9.3.2, followed by the data analysis procedure in Sub-section 9.3.3.

9.3.1 Study goal and research questions

The first study was of exploratory nature. Its main goal was to develop an understanding of approaches taken and challenges encountered by master’s students who had identified architectural patterns in an OSS product as part of courses on Software Patterns and Software Architecture. Based on these insights, the researcher developed an initial process, which consists of a set of procedural guidelines that may help practitioners to identify architectural patterns.

Whereas the main goal of the research presented in this chapter was expressed by the general research question (following Punch’s levels of abstraction presented in Figure 5-1) presented above, the study conducted in the first iteration of the Improvement Paradigm was designed to address the following three specific research questions.

SRQ1: What are the approaches that participants take to identify architectural patterns?

SRQ2: What challenges did participants encounter during pattern identification?

SRQ3: How can the process of identifying architectural patterns be improved?

9.3.2 Data collection

The researcher drew from three groups of participants in this phase, which are summarised in Table 9-1. The researcher decided to gather data from master’s students for the following reasons. Firstly, identifying patterns is not a common activity in practice. Therefore, it would have been extremely difficult to find practitioners that could be interviewed about this activity. Secondly, master’s students can be considered novice software developers. On the one hand, as such, they have limited
experience in software development; this implies that they would not have faced the need of identifying architectural patterns, and thereby have gained any experience in this task. In other words, the students would not have had prior exposure to this challenging task. On the other hand, master’s students (as opposed to undergraduate students) typically have some experience, either through internships or through part-time software development jobs. A third reason to collect data from master’s students is that they were available and willing to participate in this research. Getting access to professional participants in any research study can prove to be much more challenging. Section 9.7.10.4 discusses the threats to validity that arise from using students in extensive detail.

All three groups of participants had performed a pattern identification task in the context of a master’s course on Software Patterns or Software Architecture. The table below provides information about the data collection method (including the date of collection), the date that the participants studied their selected OSS project, the name of the studied OSS project, the university at which the participants were enrolled at the time of data collection, and finally the size of the teams in which the pattern identification assignment was performed. The numbers between parentheses indicate the numbers of participants from which data was collected; for instance, while team A had six members, only three team members were interviewed. These data sources are discussed in more detail below.

Table 9-1. Data sources for first study.

<table>
<thead>
<tr>
<th>Data collection method</th>
<th>Date studied</th>
<th>Team ID</th>
<th>OSS Project</th>
<th>University</th>
<th>Team size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semi-structured interviews</td>
<td>January 2009</td>
<td>A</td>
<td>JBoss</td>
<td>Univ. of Groningen</td>
<td>6 (5)</td>
</tr>
<tr>
<td>(February ’09)</td>
<td></td>
<td>B</td>
<td>Eclipse</td>
<td>Univ. of Groningen</td>
<td>6 (3)</td>
</tr>
<tr>
<td>Reflection reports</td>
<td>April 2009</td>
<td>C</td>
<td>MeDiCi</td>
<td>Univ. of Limerick</td>
<td>3</td>
</tr>
<tr>
<td>(May ’09)</td>
<td></td>
<td>D</td>
<td>Filezilla</td>
<td>Univ. of Limerick</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>E</td>
<td>HackyStat</td>
<td>Univ. of Limerick</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>F</td>
<td>ServiceMix</td>
<td>Univ. of Limerick</td>
<td>3</td>
</tr>
<tr>
<td>Author’s Experience</td>
<td>January 2007</td>
<td>G</td>
<td>Parrot</td>
<td>Univ. of Groningen</td>
<td>2 (1)</td>
</tr>
<tr>
<td>(September ’09)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
9.3.2.1 Semi-structured interviews

The researcher invited all 12 students at the University of Groningen in the Netherlands to participate in semi-structured interviews. The students had taken the course “Software Patterns” in Winter 2008-2009, given by one of the researcher’s supervisors. Most of the students had no or limited work experience. One of the course’s assignments was to identify at least five architectural patterns (as opposed to design patterns, such as “Factory”). The students had been given four weeks for the assignment. The researcher invited the students to participate after they had received their marks, so as to prevent the students giving any dishonest answers for fear of lower marking of their assignment. Eight of them agreed to cooperate. The eight interviews lasted between 30 and 60 minutes each; the interview guide used for these interviews can be found in Appendix B. All interviews were digitally recorded with the participants’ consent. Five of these interviews were with Dutch students, and were conducted in their native language, since the researcher is a native speaker. The other three interviews (with students from Germany, India and Tanzania) were done in English. The recorded interviews were transcribed by the researcher, and the interviews conducted in Dutch were translated into English by the researcher, to enable inspection by his two supervisors. In order to prevent that anything was “lost in translation”, the translated transcripts were checked against the original recordings. Interviewed students had been members of two teams, which had identified patterns in JBoss application server (team A) and Eclipse (team B). Both are mature and industry-strength OSS projects.

9.3.2.2 Reflection reports

The second data source was the experiences as reported by another group of master’s students that had performed an assignment on pattern identification. The assignment was one of three assignments in a course “Software Architecture” given by one of the researcher’s supervisors at the University of Limerick in Ireland. This course was given in the Spring semester in 2009. Students had been asked to identify at least five architectural patterns. The students had been given four weeks for this
assignment and were expected to spend 10 hours per week on this. These students were all doing a master’s course in software engineering at the University of Limerick.

9.3.2.3 Researcher’s experience
The third data source were the personal experiences of the researcher, who had done the same pattern identification assignment, in the context of the Software Patterns course at the University of Groningen in 2007. The assignment in this (first) edition of the course was to identify at least five architectural patterns, and was to be done in teams of two students, as opposed to six in the 2009 class (who were interviewed). It is worth mentioning that the researcher had been a contributor (with write access to the source code repository) to the studied OSS project (“Parrot”), and therefore had a thorough understanding of that project.

9.3.3 Data analysis
The collected data were analysed as follows. The researcher thoroughly read the transcripts of the interviews, and phrases that were considered relevant to our research questions were highlighted using different colours for different topics (“approach”, “challenge”, “suggestion”), so as to be able to quickly identify the topic of the phrase. The reflection reports were analysed in a similar fashion. As for the researcher’s experiences, he noted down a number of approaches and challenges that he had encountered in performing the assignment in 2007. These were subsequently labelled as described above.

All data were entered into a Microsoft Access database together with the source (filename and page number or timestamp for interview transcripts). This established an audit trail, which allowed the researcher and his supervisors for easy crosschecking and tracking back of the data to their sources. An audit trail is a recommended practice to establish validity in qualitative studies (Guba, 1981).

The researcher classified approaches, challenges and suggestions by labelling each one of them. For example, a certain approach may be to
“read the documentation”; this approach was labelled as “documentation”. Through this process, a set of categories emerged. For each label, a justification was provided to support the choice of the label. This initial classification was crosschecked by two of the researcher’s supervisors who were also involved in this research. Any disagreements were resolved through discussion. After the labelling step, approaches, challenges and suggestions were grouped together based on their label, to identify equivalent entries. Challenges were related to the approaches during which they were encountered.

Based on the identified approaches, challenges and suggestions for improvement, the researcher identified the key steps and formalised them in a process definition, using UML activity diagram notation. The process is designed to support practitioners in the task of Identifying Architectural Patterns in OSS products, or “IDAPO” for short. The order of the steps was based on the order that seemed to reflect the order of most interviewed participants.

The results of the research questions SRQ1, SRQ2 and SRQ3, as well as the initial version (v1) of IDAPO are presented in Section 9.4.

9.4 RESULTS OF ITERATION 1: APPROACHES, CHALLENGES, SUGGESTIONS AND INITIAL DESIGN OF IDAPO

This section presents the results of the first iteration of the Improvement Paradigm process. During this first iteration, the researcher focused on identifying approaches, challenges and suggestions for improvement in the task of identifying architectural patterns in an OSS product. Approaches are presented in Sub-section 9.4.1; challenges encountered are presented in Sub-section 9.4.2. Suggestions on how to improve this task are presented in Sub-section 9.4.3. Based on these results, a first draft of the process for Identifying Architectural Patterns in OSS products (IDAPO) is presented in Sub-section 9.4.4.
9.4.1 Approaches to identify architectural patterns in OSS

The analysis resulted in a list of 20 approaches that the students have taken. Related approaches are listed into a number of categories that emerged during data analysis, as described in Sub-section 9.3.3. Table 9-2 presents the approaches (numbered A1 to A20), organised per category. The remainder of this sub-section discusses these approaches in more detail.

Table 9-2. Approaches taken by participants to identify patterns.

<table>
<thead>
<tr>
<th>Category</th>
<th>ID</th>
<th>Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software under investigation</td>
<td>A1</td>
<td>Analyse usage point of view; what functions are provided</td>
</tr>
<tr>
<td></td>
<td>A2</td>
<td>Download, install, configure and run the software</td>
</tr>
<tr>
<td></td>
<td>A3</td>
<td>Create a test application with the project</td>
</tr>
<tr>
<td></td>
<td>A4</td>
<td>Identification of used technologies and web search to find more details on the used technologies.</td>
</tr>
<tr>
<td>Documentation</td>
<td>A5</td>
<td>Read published books about the project</td>
</tr>
<tr>
<td></td>
<td>A6</td>
<td>Browse and read project documentation</td>
</tr>
<tr>
<td></td>
<td>A7</td>
<td>Cross-check, study and analyse diagrams</td>
</tr>
<tr>
<td></td>
<td>A8</td>
<td>Verify documentation with source code</td>
</tr>
<tr>
<td>Source code and tools</td>
<td>A9</td>
<td>Browse and inspect the source code</td>
</tr>
<tr>
<td></td>
<td>A10</td>
<td>Systematically analyse source code using IDE [to identify package hierarchy]</td>
</tr>
<tr>
<td></td>
<td>A11</td>
<td>Look at component names in the source code</td>
</tr>
<tr>
<td></td>
<td>A12</td>
<td>Find certain constructs in the source using self-made tools.</td>
</tr>
<tr>
<td></td>
<td>A13</td>
<td>Use tools to reverse engineer the code and create diagrams</td>
</tr>
<tr>
<td>Components &amp; connectors</td>
<td>A14</td>
<td>Bottom up analysis of components to see how they are related to other components</td>
</tr>
<tr>
<td></td>
<td>A15</td>
<td>Look at API interfaces, class relationships and directory names</td>
</tr>
<tr>
<td></td>
<td>A16</td>
<td>Look up information on interface on web</td>
</tr>
<tr>
<td>Community</td>
<td>A17</td>
<td>Contact the community (e.g. through IRC channel or mailing list)</td>
</tr>
<tr>
<td></td>
<td>A18</td>
<td>Read forum posts, web log articles, development diary, wikis, project web site, mailing lists, discussion group, development diary</td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>A19</td>
<td>Read books and papers to find more information on architectural patterns</td>
</tr>
<tr>
<td></td>
<td>A20</td>
<td>Divide task over team members and verify findings</td>
</tr>
</tbody>
</table>

9.4.1.1 Software under investigation

The researcher identified the following approaches that consider the project under investigation as a whole. One approach was to analyse the application for its functionality and identify its use cases (A1). Another
approach taken was to download, install, configure and run the software (A2). One group of students wrote and deployed a simple application on top of the OSS project under investigation (A3). Whether this is applicable depends of course on the system under investigation; in this case it was the JBoss application server. The fourth approach in this category is to identify used technologies in order to find more information on those technologies (A4).

9.4.1.2 Documentation
The researcher identified the following approaches in the category “documentation”. Students consulted books about the published project (A5). Of course, typically books are only published on well-known and successful OSS products. An obvious strategy was to browse and read the provided documentation (A6). In addition, students would analyse, study and crosscheck diagrams (A7). The researcher considered studying diagrams to be different from studying documentation, as students explicitly distinguished the two approaches. Another approach was to verify the documentation with the source code (A8). This was labelled as “documentation” (as opposed to “source code”) since students would first study the documentation and then crosscheck that with the source code.

9.4.1.3 Source code and tools
The researcher classified the following approaches in the category “source code and tools”. Browsing and studying the source code was one reported approach (A9). Some students reported to have used IDEs (Integrated Development Environments) to systematically analyse the source code (A10). The researcher identified this separately from approach A9, since these approaches were also distinguished by several students. Furthermore, students searched for component names in the source code (A11). One student reported to have written a simple Perl script to identify certain constructs in the source code (A12). Almost all students reported to have tried to use tools to reverse engineer the source code and generate diagrams (A13). Among the tools that were used are Poseidon for UML, StarUML, Eclipse, NetBeans, CC-Rider, Columbus and AgileJ.
9.4.1.4 Components and connectors

The researcher identified a few approaches that students undertook related to the components and connectors (relations among components). One student reported to have taken a bottom-up approach, starting by studying one component and to identify its relationships to other components (A14). Another approach reported was to study the API interfaces, function names (“connectors”) and directory names and how classes are related to one another (“structure”) (A15). Yet another approach was to find more information on interfaces on the web (A16).

9.4.1.5 Community

The students used the following two approaches related to the project’s community. Contacting the community through the project’s IRC (Internet Relay Chat) channel was reported to be very helpful (A17). Besides this, students also looked at forum posts, web log articles about the project, and one student reported to have read a development diary which had been written by the project’s architect (A18).

9.4.1.6 Miscellaneous

The researcher identified two approaches that did not seem to fit in any other category, which is why these were classified as “miscellaneous”. The first is to read books and papers to acquire more understanding of different architectural patterns (A19). Students also divided the task over the team members to compare and verify their findings through discussion (A20).

9.4.2 Challenges in identifying architectural patterns

The researcher identified 29 different challenges that students encountered during the assignment, which are listed in Table 9-3. The identified challenges were grouped according to the respective approaches (as identified in the previous section). It should be noted that not all approaches have associated challenges. This grouping helps to gain insight in what problems may arise as a result of taking a certain approach. Such understanding can assist in defining a set of guidelines to make
pattern identification a more straightforward process. The remainder of this section discusses these challenges per approach in more detail.

9.4.2.1 **Approach A1: challenges**

Two challenges were identified for approach A1. Firstly, students reported their unfamiliarity with the domain to be a hindrance (C1). Secondly, one student reported that it was difficult to understand the software since it was not straightforward to run the software (C2); the student had therefore trouble to understand its purpose.

9.4.2.2 **Approach A2: challenges**

Five challenges were identified for approach A2. Firstly, some students encountered build errors (C3), which caused them to manually fix the build configuration. Another challenge was that some students did not have any experience or knowledge on checking out source code from the version control system (C4). Unavailability of source code (due to missing or broken links) was another challenge (C5). One student reported that the compilation process “was a nightmare” (C6); this was caused by the many dependencies on other subsystems that had to be downloaded in addition.

9.4.2.3 **Approach A3: challenges**

Two challenges were identified for approach A3. Firstly, students experienced a lack of experience in building and deploying applications (C7) and a lack of time to set up the application (C8).

9.4.2.4 **Approach A6: challenges**

The researcher identified the following challenges related to approach A6. Firstly, some students reported that the documentation assumes a rather thorough knowledge of used technologies (C9). A general lack of documentation was also reported to be an obstacle (C10). Interestingly, one student reported that there was *too much* documentation available from many different sources, and that it was challenging to find your way through various types of documentation (C11). Furthermore, it was reported that there was no overview documentation and the available documentation was of the wrong type. In other words, the target audience
of the documentation was different (C12). Another reported challenge was the uncertainty whether the available documentation was up to date for the current version of the software (C13).

**9.4.2.5 Approach A7: challenges**
The following challenges were reported related to approach A7. Firstly, students felt that there was a general lack of diagrams (C14). Students were also uncertain whether the available diagrams were still up to date and relevant for the current version of the software (C15). Furthermore, students reported that the diagrams that were available were useless and did not provide any insight into the software’s architecture (C16). Besides this, one student (C17) reported that no standards (such as UML) were used for the diagrams.

**9.4.2.6 Approach A9: challenges**
The researcher identified the following challenges relating to approach A9. Firstly, one student reported the hierarchy of the source code directory was counter-intuitive for someone with little architecting experience (C18). Another reported challenge was a large number of source code files (C19). Students also reported that manually browsing the source code for patterns was “tricky” and very time consuming (C20). It was reported that the code was not very well documented (C21), which made it more difficult to understand the source code.

**9.4.2.7 Approach A13: challenges**
A variety of challenges were encountered as a result of attempts to use tools to reverse engineer the source code. Firstly, students reported a lack of suitable tools (C22). It was also reported that the generated output (such as diagrams) was unreadable and therefore useless (C23). Students also reported that the tool they intended to use was not suitable for the programming language used for the software (which was mostly Java and C++) (C24). Also, students reported that learning to use the tool required much effort (C25). Furthermore, some students reported that the tools they had tried simply did not work or continuously got frozen, which made their use impossible (C26). Lastly, another problem was that the
tool the students intended to use could not handle the directory structure of the source code (C27).

9.4.2.8 Approach A17: challenges
Students contacted the community to ask for help and feedback. Two challenges were identified in this approach. Firstly, the community’s response was not considered to be very helpful (C28). For instance, one student reported that another student (from another institute) had received this reply: “the codebase is the documentation”. Other students reported that the community had not replied (yet) on their request for feedback (C29).
Table 9-3. Challenges encountered by participants during the identification of architectural patterns in OSS products.

<table>
<thead>
<tr>
<th>A.ID</th>
<th>Approach</th>
<th>C.ID</th>
<th>Challenge</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>Analyse usage point of view, use cases, software functionality</td>
<td>C1</td>
<td>Unfamiliarity with domain</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C2</td>
<td>Understanding software was difficult as it could not easily be run as e.g. an office application</td>
</tr>
<tr>
<td>A2</td>
<td>Download, configure, install and run the software</td>
<td>C3</td>
<td>Errors during building the software</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C4</td>
<td>Unfamiliarity with checking out source code</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C5</td>
<td>No access to source code due to broken hyperlinks</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C6</td>
<td>Compilation is very complex due to many dependencies</td>
</tr>
<tr>
<td>A3</td>
<td>Create a test application with the project</td>
<td>C7</td>
<td>Lack of experience in building/deploying (web) applications</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C8</td>
<td>Lack of time to set up a test application</td>
</tr>
<tr>
<td>A6</td>
<td>Browse and read project documentation</td>
<td>C9</td>
<td>Documentation assumes knowledge of used technologies</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C10</td>
<td>Lack of documentation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C11</td>
<td>Too much documentation; difficult to find your way in the large amount of documentation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C12</td>
<td>No overview documentation, the documentation was not relevant or for other types of users</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C13</td>
<td>Not sure if documentation is still up to date.</td>
</tr>
<tr>
<td>A7</td>
<td>Study, analyse and cross-check diagrams</td>
<td>C14</td>
<td>Lack of diagrams</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C15</td>
<td>Not sure if diagrams are still up to date.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C16</td>
<td>Available diagrams are useless.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C17</td>
<td>No standard is used for diagrams (such as UML)</td>
</tr>
<tr>
<td>A9</td>
<td>Browse, study, inspect source code</td>
<td>C18</td>
<td>Hierarchy of source code directory organisation is counter intuitive</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C19</td>
<td>Amount of source code files is large</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C20</td>
<td>Manually browsing source code is tricky and time consuming</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C21</td>
<td>Code comments are not clear</td>
</tr>
<tr>
<td>A13</td>
<td>Using tools to reverse engineer source code, create diagrams</td>
<td>C22</td>
<td>Lack of suitable tools.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C23</td>
<td>Tool output (e.g. generated diagrams) is useless or unreadable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C24</td>
<td>Tools are not suitable for the programming language</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C25</td>
<td>Tools require too much effort to learn to use</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C26</td>
<td>Tools failed (generation of diagram failed, tools froze).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C27</td>
<td>Tools cannot handle source code directory structure.</td>
</tr>
<tr>
<td>A17</td>
<td>Contact community (IRC, mailing lists)</td>
<td>C28</td>
<td>Community response is not helpful</td>
</tr>
<tr>
<td></td>
<td></td>
<td>C29</td>
<td>No response from community</td>
</tr>
</tbody>
</table>
9.4.3 Suggestions for improvement

The researcher asked the eight interviewees to give suggestions about improving the pattern identification process. In formulating the questions, the researcher asked specifically for ideas that could help in designing a tool or a pattern identification method. The results from the analysis of the responses to this question are presented in Table 9-4. The suggestions (numbered S1 to S16) were categorised in a similar way as the approaches in Sub-section 9.4.1. The suggestions are discussed in more detail below.

Table 9-4. Suggestions of participants for improving the pattern identification process.

<table>
<thead>
<tr>
<th>Category</th>
<th>ID</th>
<th>Suggestion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software under investigation</td>
<td>S1</td>
<td>Handbook describing different types of systems, and typical patterns used in each of those types of systems, e.g. Web applications often use client/server pattern</td>
</tr>
<tr>
<td></td>
<td>S2</td>
<td>Use similar systems as a starting point to identify potentially used patterns</td>
</tr>
<tr>
<td></td>
<td>S3</td>
<td>Use similar systems as a starting point to identify probable locations to find patterns</td>
</tr>
<tr>
<td>Documentation</td>
<td>S4</td>
<td>Create high-level UML diagrams, identify patterns manually</td>
</tr>
<tr>
<td></td>
<td>S5</td>
<td>Increase knowledge of patterns, and look for relevant patterns</td>
</tr>
<tr>
<td></td>
<td>S6</td>
<td>Reverse engineer code to get UML diagrams</td>
</tr>
<tr>
<td></td>
<td>S7</td>
<td>Acquire entire view of architecture, sequence diagrams</td>
</tr>
<tr>
<td>Source code and tools</td>
<td>S8</td>
<td>Find location of definition of classes named after patterns (e.g. “Broker”)</td>
</tr>
<tr>
<td></td>
<td>S9</td>
<td>Look for names of patterns in the source code</td>
</tr>
<tr>
<td></td>
<td>S10</td>
<td>Standardise names of participants of patterns, connect terms to participants</td>
</tr>
<tr>
<td></td>
<td>S11</td>
<td>Draw suspected structure of program and let tool identify patterns.</td>
</tr>
<tr>
<td></td>
<td>S12</td>
<td>Parse relevant parts of the code (such as class declarations)</td>
</tr>
<tr>
<td></td>
<td>S13</td>
<td>Let tool give suggestions of possible patterns based on knowledge/database of patterns</td>
</tr>
<tr>
<td>Components &amp; connectors</td>
<td>S14</td>
<td>Identify what the components of the software are; what source file(s) makes up a component</td>
</tr>
<tr>
<td></td>
<td>S15</td>
<td>Identify (both compile-time and run-time) relationships and dependencies of components</td>
</tr>
<tr>
<td></td>
<td>S16</td>
<td>Identify component communication, interaction and methods’ returned values</td>
</tr>
</tbody>
</table>
9.4.3.1 **Software under investigation**

One student suggested the development and use of a “handbook”, which lists different types of software and common patterns per type of software (S1). This matches the vision of Grady Booch in his efforts to create a “handbook of software architecture” (Booch, 2010). Harrison and Avgeriou have classified 47 architectures from Booch’s repository into seven domains and identified 110 patterns (Harrison and Avgeriou, 2008). Such a repository could then be consulted as part of the pattern identification process, so that a practitioner knows what to look for. Looking at other similar software as a starting point to identify potentially used patterns was also suggested (S2). Similarly, it was suggested to look at similar software to identify likely locations for those patterns identified through suggestions S1 and S2 (S3).

9.4.3.2 **Documentation**

The participants indicated the need for creating high-level UML diagrams that can be used for manual pattern identification (S4). Increasing knowledge of patterns and identifying relevant patterns for the software under investigation was also deemed helpful (S5). Furthermore, students suggested generating UML diagrams from reverse engineering the source code (S6). Students also indicated it would be helpful to acquire a view on the entire architecture (S7). This came from their experience that the software they investigated only provided diagrams of various sub-systems, but not the architecture in its entirety. Sequence diagrams were mentioned explicitly, as they provide the interaction between various sub-systems.

9.4.3.3 **Source code and tools**

When inspecting the source code, the following ideas were considered useful. Finding the locations of definitions of classes that are named after patterns (such as “Broker”), or components of patterns (such as the “Model” in Model-View-Controller (MVC)) was mentioned as a useful way to study the source code (S8). Generally looking for pattern names in the source code was also mentioned as a fruitful task (S9). Standardising patterns’ names, identifying the terms used for patterns (or pattern components) was also suggested, since different names are sometimes
used for the same patterns (S10). Using some workbench of some sort with an editor, a user could draw the suspected structure of a program and let the tool identify any patterns based on the input (S11). Another idea was to have a tool to parse only potentially relevant parts of the code, such as class declarations and ignoring the rest (S12). Furthermore, it was suggested to have a tool that suggests potential patterns based on the tool’s knowledge (such as a database) of pattern definitions (S13).

9.4.3.4 Components and connectors

Students suggested that it was important to identify the components in the software under investigation, by looking at what source files contain the component’s implementation (S14). Furthermore, students think it is fruitful to identify relationships and dependencies of components (S15). It was also suggested to look at the communication and interaction between components as well as methods’ returned values (S16).

9.4.4 Initial design of IDAPO

Since architectural pattern identification cannot be fully automated, the researcher argues that it is important to provide a procedural set of guidelines that may help practitioners to identify patterns. Based on the insights into approaches, challenges and suggestions presented in the previous three Sub-sections, the researcher defined a process which consists of a number of steps that can be taken to identify patterns. The process is based on empirical findings; the steps that are defined are based on the participants’ approaches and enriched with the participants’ suggestions. As the various steps were defined, the researcher considered the various challenges in order to provide guidance to the process’ users. References are provided to approaches (A1 to A20) and suggestions (S1 to S16) from the previous sub-sections in the descriptions of the various steps of the process so as to indicate the relation to those empirically identified approaches and suggestions. It must be noted that the researcher did not include any use of tools in the process, which is why no references are made to approach A13 and suggestions S10-S12. The researcher emphasises that the effectiveness of this process is heavily
dependent on the “input”, such as the quality of the source code, the available documentation and the level of experience of the practitioner. The initial version of IDAPO is shown in Figure 9-3, using UML activity diagram notation.

**Figure 9-3. Initial design of IDAPO.**

The researcher asserts the importance of a tool that acts as a repository for architectures and their contained patterns. Examples are the handbook created by Grady Booch (Booch, 2010), the Open Pattern Repository (van Heesch, 2010) and PAKME (Ali Babar and Gorton, 2007). The process is discussed in detail in the following sub-sections.

**9.4.4.1 Identify type and domain of software**

The first step is to identify the type and domain of the software under investigation. The output of this step is to get an overview of the product’s functionality (A1) and its domain (for instance, a content management system (CMS) or an application server). Installing and running the software may be helpful in this step (A2).
9.4.4.2  Identify candidate patterns

The next step is to use the information from step one to consult a repository (S1) that contains information about patterns used and their potential location by various system types. Since the repository may not contain sufficient information initially, it would be advised to consult documentation of similar systems (for instance, other CMSs) (S2, S3). The output of this step is a list of potentially (or likely) used patterns and their possible location.

9.4.4.3  Read literature to learn about patterns

It is not expected that a practitioner would have knowledge of all possible patterns that have been published. Therefore, a practitioner should consult the literature to acquire more information about the potentially used patterns from step two (A19, S5). The result of this step is an increased understanding of the potentially used patterns.

9.4.4.4  Identify used technologies

The next step is to identify used technologies of the software (A4). An understanding of this may help in the identification process. For instance, if the software uses certain frameworks, such as the Struts framework for web-based Java applications, this may hint the presence of certain patterns (such as MVC).

9.4.4.5  Study used technologies

If the practitioner does not have sufficient knowledge of the used technologies (A4), the next step is to find more information about this, which will result in a better understanding of the used technologies, and give more insight in potentially used patterns.

9.4.4.6  Study documentation

If there is sufficient documentation available, the researcher proposes to study this first as opposed to studying the source code (A5-A7, S5-S7), which is much more tedious. Based on this step, the practitioner may already be able to identify some patterns.
9.4.4.7 Study source code

If there is no documentation available, the practitioner will have to resort to studying the source code (A9, A10, S7, S8). Also, the documentation may be verified by studying the implementation (A8). Furthermore, the source code may contain comments that hint on the use of certain patterns.

9.4.4.8 Study components and connectors

Based on the documentation and source code, a practitioner may have an initial idea of the structure of the software and the various connectors. In order to improve this insight, the researcher argues to have a number of iterations to go back to the documentation and source code, to verify or adjust these insights.

9.4.4.9 Identify patterns

Once a practitioner has gained good insight into the components and connectors of the software, the next step is to identify patterns. During this step, literature may be consulted for additional information about the patterns (A19).

9.4.4.10 Contact community for feedback

After finding the patterns, the researcher proposes it is valuable to contact the studied OSS project’s community to ask for feedback on the findings (A17). Although the community could of course be contacted in an earlier phase for asking additional information (in case of lacking documentation), the researcher argues that providing the community with something that they can comment on may yield more fruitful responses. Community members typically have limited time available and wish to spend that time as efficient as possible.

9.4.4.11 Validate identified patterns

The next step is then to validate the identified patterns. If the practitioner needs more information, the literature may (once again) be consulted (A17). The output of this step is a list of confirmed patterns and/or a list of unconfirmed patterns.
9.4.4.12 Register usage of patterns

If the presence of the patterns can be confirmed, the next step is to register the patterns and their variants in the repository. Otherwise, it is suggested to go back to study the literature on patterns in order to gain a better understanding of patterns.

9.4.5 Limitations of this study

The researcher is aware of a few limitations of this study. Firstly, the researcher single-handedly conducted, transcribed and translated the interviews, without any crosschecking of other researchers. This may have resulted in loss of information. However, during transcription of the interviews, much care was given to record as much information as possible. In order to ensure that the translation did not introduce errors, the researcher checked the translated transcripts against the recordings.

A second limitation of this research is that it is unlikely that the researcher has captured all approaches and challenges for identifying architectural patterns. However, this field of study is still very young, and the researcher therefore feels that the exploratory nature of this study is a useful contribution.

A third limitation of this study is the potential bias of the third data source (the researcher’s experience), as his results were noted down after collecting data from the other participants. However, the goal of the research was to gain insights into approaches and challenges of pattern identification, and the researcher feels that this does not influence the validity of the findings in this exploratory research.

Finally, creating a well-fitting classification of approaches and challenges is not straightforward, and subjective by nature. Sometimes an approach may fall into two different categories, for instance, verifying documentation with source code (approach A8). On the one hand this could be classified as “documentation”; on the other hand this may be classified as “source code”. The researcher resolved any of these
disagreements through discussion with two of his supervisors, who were also involved in this research.

9.5 DESIGN OF ITERATION 2: IMPROVE UNDERSTANDING, CROSS-CHECKING AND ENHANCEMENT OF IDAPO

This sub-section presents the design of the study conducted in the second iteration of the Improvement Paradigm. Its goal is presented in Sub-section 9.5.1, the data collection procedure is discussed in Sub-section 9.5.2, followed by the data analysis procedure in Sub-section 9.5.3.

9.5.1 Study goal

The second study was conducted to improve the researcher’s understanding of how to identify architectural patterns. In order to validate the different steps of the initial process as well as to enhance the process, the researcher gathered additional data from another group of master’s students who had performed a pattern identification assignment in the context of a master’s course on Software Patterns (at the University of Groningen). By gathering data from another set of students, the researcher performed triangulation across data sources, which is a validity procedure to converge among multiple and different sources of information (Creswell and Miller, 2000).

9.5.2 Data collection

In order to collect more data, the researcher invited all 12 students who were enrolled in the course for a semi-structured interview. As for iteration 1, the students were invited after they had received their grades, in order to prevent the students from giving dishonest answers. Ten students chose to participate. The participants were not shown the initial process in order to prevent getting only confirmatory answers. Instead, the researcher asked the students about the steps taken, their usefulness, what information they had been looking for, obstacles they had encountered, and their “lessons learned”, i.e., what steps they would and would not take again. The interviews were digitally recorded with the
participants’ consent. Dutch students were interviewed in their native language. The other students were interviewed in English. The interviews lasted 60 minutes on average. All interviews were transcribed verbatim by the researcher. The Dutch transcriptions were translated into English to allow the researcher’s supervisors in the data analysis.

9.5.3 Data analysis

The data were analysed in a similar fashion as in iteration 1, reported in 9.3.3. The researcher systematically extracted information about the steps that the students had taken and recorded them in a spreadsheet. The researcher compared the steps to the activities of the initial version of IDAPO. The researcher focused primarily on the steps that students had considered to be useful; for instance, many students considered the use of reverse engineering tools to be a waste of time. The improvements made and the new version (v2) of IDAPO are presented in Section 9.6.

9.6 RESULTS OF ITERATION 2: ENHANCEMENT OF IDAPO

The initial process definition of IDAPO (presented in the previous section) was based on insights gained from master’s students who had performed a pattern identification assignment. In order to validate the process steps, data was collected from an additional group of 10 master’s students, who had also performed the same pattern identification task.

The researcher found that the steps taken by the students mostly corroborated the activities of our initial process. The researcher also found reason to make a number of changes based on new insights gathered from the 10 interviews. While Figure 9-4 presents the enhanced process in more detail, in this section the changes are briefly summarised.

The researcher realised that an incremental accumulation of information, such as type and domain of the product as well as implementation technologies used, could be a useful way to identify potentially used patterns, which the researcher refers to as candidate patterns. Therefore, the researcher swapped the order of steps (4, 5) with
steps (2, 3) in the initial version presented in Figure 9-3. A second change that was made was to enrich the process with a data flow between the steps, which describes the different pieces of information that can be gathered as a user follows the steps as well as which steps use this information. Thirdly, the researcher used the Business Process Modelling Language (BPMN) to define the process to replace the UML activity diagram notation that was used before. This allowed the researcher to more clearly express the different steps of the process.
Figure 9-4. Enhanced version of IDAPO (v2) in Business Process Modelling Language (BPMN) notation.
9.7 ITERATION 3: EMPIRICAL EVALUATION OF IDAPO

The researcher conducted a third iteration of the understand-assess-package cycle of the Improvement Paradigm. Following the call by Falessi et al. (2010) who argue that new techniques should be empirically evaluated in order to improve the state of practice in software architecture, the researcher designed the third iteration to acquire an evidence-based understanding of the usefulness of IDAPO. Usefulness is measured in terms of the number of patterns that are identified. The evaluation of IDAPO was performed by means of an experiment, and allowed the researcher to assess the process in order to determine potential improvements. The third step of the Improvement Paradigm process would be to package the improvements by documenting them. However, the researcher did not perform this last step, which is left for future work.

The remainder of this section presents the design of the experiment; this presentation is based on the reporting guidelines for experiments proposed by Jedlitschka et al. (2008).

9.7.1 Study goals and hypotheses

The researcher defined three goals for this experiment. Firstly, the researcher was interested in whether using IDAPO helps to identify more patterns. The researcher argues that the task of correctly identifying architectural patterns depends on practitioners’ expertise and experience; if the use of IDAPO results in more identified patterns, this expertise is important to assess their correctness. However, not all practitioners have extensive expertise to draw from. In order to be able to more precisely evaluate the usefulness of IDAPO, the second goal of this experiment was to measure the output in terms of two standard measures: precision and recall (Olson and Delen, 2008). These terms have been extensively discussed in Chapter 6 (Section 6.2.1.1). The third goal is to investigate to what extent IDAPO supports the identification of candidate patterns based on information gathered in the first three steps of IDAPO. Candidate patterns are patterns that are potentially present, based on
insights gained from studying the information gathered in the first three steps of the process. To investigate these three goals, the researcher defined six hypotheses, which are discussed next.

To address the first goal, the researcher decided to simply count the number of identified patterns, disregarding whether the patterns are correct or not. IDAPO describes the steps to take, and the information required to identify patterns. Hence the first hypothesis:

\[ H_{01}: \text{Using IDAPO does not change the number of identified patterns}. \]

For all hypotheses, the researcher implies a comparison to the number of patterns identified when not using IDAPO.

Besides looking at the number of identified patterns tested in \( H_{01} \), it is also useful to use standard measures based on a confusion matrix, namely precision and recall (Olson and Delen, 2008). Precision and recall were discussed in detail in Chapter 6. Hence, the researcher defined hypotheses \( H_{02} \) and \( H_{03} \).

\[ H_{02}: \text{Using IDAPO does not change the precision of identified patterns}. \]

\[ H_{03}: \text{Using IDAPO does not change the recall of identified patterns}. \]

The process emphasises a step-wise, incremental approach to gather information in a systematic way. In particular, the researcher was interested in the candidate patterns based on the first few steps of the process. In order to test this idea, the researcher defined hypothesis \( H_{04} \):

\[ H_{04}: \text{Using IDAPO does not change the number of candidate patterns}. \]

Likewise, the researcher decided to also test precision and recall rates for the candidate patterns. Hence, the researcher defined to \( H_{05} \) and \( H_{06} \):

\[ H_{05}: \text{Using IDAPO does not change the precision of candidate patterns}. \]

\[ H_{06}: \text{Using IDAPO does not change the recall of candidate patterns}. \]
For each hypothesis, the researcher implies an alternative hypothesis $H_{an}$ ($n=1$ to 6) that states that the use of IDAPO does result in a higher number of (candidate) patterns.

### 9.7.2 Participants and training

The researcher invited 24 master’s students who were enrolled in a course on Software Patterns at the University of Groningen, to participate in the experiment. The course was given by one of the researcher’s supervisors. Participation was not compulsory, but students were advised to participate, as one of the upcoming course assignments would also be to identify patterns in an OSS product in order to perform a pattern-based architecture review (Harrison and Avgeriou, 2011); the embedded study with the students was therefore integrated with the course (Carver et al., 2010). Fourteen students chose to participate. Table 9-5 presents demographic information of the participants.

**Table 9-5. Participants of the experiment.**

<table>
<thead>
<tr>
<th>Group</th>
<th>ID</th>
<th>Age</th>
<th>Work experience</th>
<th>Degrees</th>
<th>Nationality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control</td>
<td>P1</td>
<td>24</td>
<td>3½ years, developer</td>
<td>B. (CS)</td>
<td>Netherlands</td>
</tr>
<tr>
<td></td>
<td>P2</td>
<td>27</td>
<td>—</td>
<td>B. (AIM)</td>
<td>Greece</td>
</tr>
<tr>
<td></td>
<td>P3</td>
<td>28</td>
<td>¼ year, developer</td>
<td>B. (CS)</td>
<td>Argentina</td>
</tr>
<tr>
<td></td>
<td>P4</td>
<td>28</td>
<td>—</td>
<td>B. (BI, CS)</td>
<td>Netherlands</td>
</tr>
<tr>
<td></td>
<td>P5</td>
<td>25</td>
<td>1 year, web developer</td>
<td>B. (CS)</td>
<td>Greece</td>
</tr>
<tr>
<td></td>
<td>P6</td>
<td>29</td>
<td>5 years, developer</td>
<td>B. (CS); M. (Psy)</td>
<td>Belgium</td>
</tr>
<tr>
<td></td>
<td>P7</td>
<td>25</td>
<td>—</td>
<td>B. (BI)</td>
<td>South Africa</td>
</tr>
<tr>
<td>Treatment</td>
<td>P8</td>
<td>27</td>
<td>2 years, developer</td>
<td>B. (CS)</td>
<td>Netherlands</td>
</tr>
<tr>
<td></td>
<td>P9</td>
<td>25</td>
<td>1 year, web developer</td>
<td>B. (CS)</td>
<td>South Africa</td>
</tr>
<tr>
<td></td>
<td>P10</td>
<td>23</td>
<td>2 years, web developer</td>
<td>B. (CS)</td>
<td>Netherlands</td>
</tr>
<tr>
<td></td>
<td>P11</td>
<td>24</td>
<td>5 years, OSS developer</td>
<td>B. (CS)</td>
<td>Netherlands</td>
</tr>
<tr>
<td></td>
<td>P12</td>
<td>25</td>
<td>2 years, web developer</td>
<td>B. (CS)</td>
<td>Spain</td>
</tr>
<tr>
<td></td>
<td>P13</td>
<td>22</td>
<td>—</td>
<td>—</td>
<td>Netherlands</td>
</tr>
<tr>
<td></td>
<td>P14</td>
<td>21</td>
<td>—</td>
<td>B. (CS)</td>
<td>Netherlands</td>
</tr>
</tbody>
</table>

Section 9.7.4 discusses the assignment procedure to the groups. The average age of the control group was almost 24, whereas the average age of the treatment group was approximately 26½. Note that work experience should be interpreted as part-time jobs. One participant (P11)
actively contributed to a small OSS project. All but one participant (P13) had finished a bachelor’s (B) degree in computer science (CS), bioinformatics (BI) or applied informatics and multimedia (AIM). One participant (P6) also had a master’s (M) degree in psychology (Psy). Most of them had varying levels of expertise in different topics, as listed in Table 9-6, e.g., three participants assessed themselves as having advanced knowledge of software engineering.

When the researcher conducted the experiment, the students had attended six two-hour lectures of the eight-week course on Software Patterns. All students also had followed a course on Software Architecture. The data presented in Table 9-5 and Table 9-6 were gathered through a pre-study questionnaire the day before the experiment.

### Table 9-6. Participants’ self-assessed levels of expertise.

<table>
<thead>
<tr>
<th>Topic</th>
<th>None</th>
<th>Beginner</th>
<th>Interm.</th>
<th>Advanced</th>
<th>Expert</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software engineering</td>
<td>0</td>
<td>5</td>
<td>6</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Software architecture</td>
<td>0</td>
<td>6</td>
<td>7</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>“Gang of Four” design patterns</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Architectural patterns</td>
<td>0</td>
<td>10</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Development process of OSS</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Experience w. integrating COTS</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

#### 9.7.3 Task and materials

The task given to the participants was to identify as many architectural patterns in a specified OSS project as possible: the JBoss application server. This project was selected for three reasons. Firstly, it is an industry-strength system (no ‘toy’ project), which is widely used in industry. Secondly, the researcher expected that the participants would be able to find sufficient information about this product in the limited available time, since it is well known and extensive documentation is available. Thirdly, the researcher already had insight into the architectural patterns used in this product, which would be needed as a marking scheme for assessing the number of correctly identified patterns as well as the precision and recall. Participants in both groups were handed out the
assignment form. The treatment group was given two additional instruments: (1) the enhanced version (v2) of IDAPO, accompanied by a description of each step; and (2) a simple spreadsheet template to record information found in each step. Additionally, the participants had access to the five volumes of the Pattern-Oriented Software Architecture (POSA) series of books (e.g., (Buschmann et al., 1996)), which list various software patterns.

9.7.4 Experiment design

The experiment design was a between-subjects design, to compare results from a control group and a treatment group. Based on previous experience in conducting research with students, the researcher expected that the participants would have varying levels of experience and expertise. Since this would have constituted a threat to the outcome of the experiment, the researcher decided to non-randomly assign participants to the control and treatment groups. Hence, this experiment was a quasi-experiment (Kitchenham et al., 2002; Kampenes et al., 2007). Eight participants had indicated to have other course obligations in either the morning or afternoon of the day of the experiment; based on this information, three participants were assigned to the control group, and five were assigned to the treatment group. Based on the information about work experience gathered in the pre-study questionnaire, the researcher assigned the remaining six students, resulting in two equally sized and approximately equivalent groups (Table 9-5).

The treatment, or independent variable manipulated by this study is the reference process, with one treatment: IDAPO is provided, and one control: IDAPO is not provided. The dependent variable is the number of architecture patterns identified by the participants using and not using the process.

9.7.5 Experiment procedure

The experiment was conducted in two sessions. The control group performed the task in the morning session, and the treatment group
(provided with IDAPO) was invited for the afternoon session. This order ensured that the control group did not see IDAPO (to prevent the diffusion or imitation of treatment threat (Wohlin et al., 2000)). In both sessions, the researcher gave a brief introduction (15 min) to explain the background and motivation of identifying patterns. For the treatment group, the researcher also explained the different steps of IDAPO. Both groups were given two hours for this task. One participant in the control group had to leave 30 minutes early due to other course obligations (P1). After the two hours, participants were asked to fill out a post-study questionnaire; the researcher designed separate post-questionnaires for the two groups, as only the treatment group could be asked about their experiences with IDAPO.

9.7.6 Establishing a set of trusted patterns

In order to be able to determine precision and recall measures, the researcher needed to compare the findings to a certain set of “correct” patterns of which the researcher is confident that they are present in the product. In order to establish such a trusted subset of patterns, three different sources were used. Firstly, the researcher used a research report that presents an analysis of the JBoss architecture (v.2.2.4, 2002) (Liu, 2002). Secondly, the researcher used a technical report (from 2005) that reports on the architecture recovery of JBoss (Salehie et al., 2005). Thirdly, the researcher used a report from a previous group that had identified patterns in JBoss in the context of the 2009 edition of the Software Patterns course (team A in Table 9-1); one of its authors had extensive professional experience as an administrator of JBoss. Table 9-7 lists patterns identified by the different sources, as well the patterns that were included in the trusted subset.
Table 9-7. Derivation of a trusted subset of patterns.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Liu</th>
<th>Salehie et al.</th>
<th>Report 2009</th>
<th>Trusted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microkernel</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Layers</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
<td>Yes</td>
</tr>
<tr>
<td>Pipes &amp; Filters</td>
<td>Yes</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Broker</td>
<td>Yes</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Dynamic proxy</td>
<td>Yes</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Proxy</td>
<td>Yes</td>
<td>Yes</td>
<td>-</td>
<td>Yes</td>
</tr>
<tr>
<td>Interceptor</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Client/server</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Active repository</td>
<td>-</td>
<td>-</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Factory</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

This selection was made based on the reports, which described the patterns and their location in JBoss, as well as the level of confidence that the researcher had in the presence of these patterns. During the selection, the researcher also considered that the different sources have studied different versions of JBoss. The researcher could not find sufficient justification to include the Pipes-Filters and the Factory patterns. The column ‘Trusted’ indicates which patterns are included in the trusted subset. All identified patterns (for both control and treatment group) were listed in a spreadsheet. In order to calculate precision and control measures, only those patterns that were listed in the trusted list (Table 9-7) were counted.

9.7.7 Descriptive statistics

Table 9-8 presents the descriptive statistics of the results. The number of identified patterns by the control and treatment group were counted as a whole. The first three columns list the results when counting all patterns, disregarding their correctness; column 1 lists the total number of patterns of the control group (18); column 2 lists the total number of candidate patterns identified by the treatment group (‘Treatment candidate’, 36), and column 3 lists the total number of identified patterns (as output of step 9 in the process, see) listed by the treatment group (‘Treatment final’, 16).
Table 9-8. Number of patterns per group, mean and standard deviation. Columns 1-3 consider all patterns identified; columns 4-6 only consider the trusted patterns.

<table>
<thead>
<tr>
<th></th>
<th>Counting all patterns</th>
<th>Counting trusted patterns only</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(1) Control (2) Treatment candidate (3) Treatment final</td>
<td>(4) Control (5) Treatment candidate (6) Treatment final</td>
</tr>
<tr>
<td>Total</td>
<td>18 36 16</td>
<td>10 21 10</td>
</tr>
<tr>
<td>Mean</td>
<td>2.6 5.1 2.3</td>
<td>1.4 3.0 1.4</td>
</tr>
<tr>
<td>Std. dev</td>
<td>2.1 2.3 2.4</td>
<td>0.9 1.1 1.8</td>
</tr>
</tbody>
</table>

Columns 4-6 show the results similarly as column 1-3, but only taking the trusted patterns into account (resulting in 10, 21 and 10 patterns, respectively). When counting trusted patterns only, there is no difference between the control group and the final results of the treatment group. The treatment group as a whole identified 21 candidate patterns, which suggests the treatment group was on the right track. Figure 9-5 shows boxplot diagrams for the results presented in columns 1-6.

Figure 9-5. Distribution of numbers of identified patterns by the control group, treatment group (candidate and final). Boxplots 1-3: counting all patterns, corresponding to columns 1-3 in Table 9-8. Boxplots 4-6: trusted patterns only, corresponding to columns 4-6 in Table 9-8.

Table 9-9 presents the mean and standard deviation values of the precision and recall rates, calculated for the control group and the treatment group. For the latter, the researcher calculated precision and recall both for the candidate results and the final results. Table 9-9 shows that the average precision of the control group is 0.56 with an average recall of only 0.18. This suggests that about half of the control group’s
patterns are correct, but that (on average) less than 20% of the trusted patterns were identified.

Table 9-9. Precision and recall for control, treatment candidate and treatment final results.

<table>
<thead>
<tr>
<th></th>
<th>Control</th>
<th>Treatment</th>
<th>Candidate patterns</th>
<th>Final</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Precision</td>
<td>Recall</td>
<td>Precision</td>
<td>Recall</td>
</tr>
<tr>
<td>Mean</td>
<td>0.56</td>
<td>0.18</td>
<td>0.62</td>
<td>0.38</td>
</tr>
<tr>
<td>Std. dev.</td>
<td>0.32</td>
<td>0.11</td>
<td>0.21</td>
<td>0.13</td>
</tr>
</tbody>
</table>

The candidate results of the treatment group score better, with a precision of 0.62 at a recall rate of 0.37, suggesting that (on average) the treatment group identified more correct candidate patterns. However, when looking at the final results of the treatment group precision is only 0.30 at a recall of 0.18, worse than the control group. The relative high values for the standard deviations of precision (0.35) and recall (0.23) for the treatment group’s final results suggest a large variation among participants. The researcher found that three participants in the treatment group did not list any “final” patterns (as opposed to one participant in the control group).

9.7.8 Results of statistical analysis

The researcher performed statistical analyses on the number of identified patterns and the calculated precision and recall rates to test the six hypotheses. The assumptions underlying parametric tests such as the t-test were not fulfilled (Hollander and Wolfe, 1999), since the data contained outliers and the researcher could not assume that the data have a normal distribution. Therefore, the researcher decided to use the Mann-Whitney U test, which is a non-parametric alternative to the t-test for two independent samples (Hollander and Wolfe, 1999). Table 9-10 lists the p-values for each of the six hypotheses. The columns ‘Candidate’ and ‘Trusted’ indicate whether the hypotheses consider the candidate patterns (of the treatment group) and whether only trusted patterns were counted, respectively. A hypothesis is rejected if the p-value is less than the
significance level of $\alpha=0.05$. The researcher used SPSS version 18 for all statistical tests.

**Table 9-10. Hypotheses and resulting p-values of the Mann-Whitney U test.**

<table>
<thead>
<tr>
<th>Hyp.</th>
<th>Variable</th>
<th>Candidate</th>
<th>Trusted</th>
<th>P-value</th>
<th>Decision</th>
</tr>
</thead>
<tbody>
<tr>
<td>H01</td>
<td>Number of identified patterns</td>
<td>No</td>
<td>No</td>
<td>0.435</td>
<td>Retain H01</td>
</tr>
<tr>
<td>H02</td>
<td>Precision of identified patterns</td>
<td>No</td>
<td>Yes</td>
<td>0.324</td>
<td>Retain H02</td>
</tr>
<tr>
<td>H03</td>
<td>Recall of identified patterns</td>
<td>No</td>
<td>Yes</td>
<td>0.597</td>
<td>Retain H03</td>
</tr>
<tr>
<td>H04</td>
<td>Number of candidate patterns</td>
<td>Yes</td>
<td>No</td>
<td>0.051</td>
<td>Retain H04</td>
</tr>
<tr>
<td>H05</td>
<td>Precision of candidate patterns</td>
<td>Yes</td>
<td>Yes</td>
<td>0.555</td>
<td>Retain H05</td>
</tr>
<tr>
<td>H06</td>
<td>Recall of candidate patterns</td>
<td>Yes</td>
<td>Yes</td>
<td>0.026</td>
<td>Reject H06</td>
</tr>
</tbody>
</table>

Table 9-10 shows that no compelling evidence to reject hypotheses H01-H05 (all p-values > $\alpha=0.05$) could be found. In other words, there was not sufficient evidence to conclude that using IDAPO resulted in a higher number of identified patterns (H01), a higher precision of identified patterns (H02), a higher recall of identified patterns (H03), a higher number of candidate patterns (H04), and a higher precision of candidate patterns (H05). With respect to H04, the researcher found that there is *some* evidence (p=0.051) that using IDAPO results in a higher number of candidate patterns, but since the p-value (0.051) is *larger* than the significance level ($\alpha=0.05$) the researcher did not reject H04. On the other hand, the researcher found evidence (p=0.026 < 0.05) to reject hypothesis H06 (‘using IDAPO does not change the recall of candidate patterns’). Together, these results suggest that IDAPO helps to improve the recall of candidate patterns.
9.7.9 Results of post-study questionnaires

The post-study questionnaire questions were rated using a five-point Likert scale, ranging from Totally Disagree (TD), to Disagree (D), Neutral (N), Agree (A) and Totally Agree (TA). The results of the post-study questionnaires of the treatment and control groups are discussed separately below.

9.7.9.1 Results of treatment group

Table 9-11 presents the results for the treatment group. Numbers indicate the number of participants that gave a certain rating, e.g., two subjects answered ‘Neutral’ on question T1.

<table>
<thead>
<tr>
<th>ID</th>
<th>Question</th>
<th>TD</th>
<th>D</th>
<th>N</th>
<th>A</th>
<th>TA</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>I followed the process step by step in the order prescribed.</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>T2</td>
<td>Identifying the type and domain of the software is helpful.</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>T3</td>
<td>Identifying the used technologies is helpful to identify patterns.</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>T4</td>
<td>The process helped me to identify patterns that I wouldn’t have found otherwise.</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>T5</td>
<td>The suggested order of steps in the process made sense.</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>T6</td>
<td>Storing information per step in the spreadsheet was useful.</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 9-11 shows that the degree to which the subjects followed IDAPO varied (T1). This suggests that participants disliked the process’ rigid order of steps, and would like to have more flexibility. The results for T2 suggest that most subjects agree that identifying the type and domain of the software is helpful. The results show that identifying used technologies (T3) was considered to be very helpful. Most participants did not think that IDAPO helped to identify patterns that could not have been identified otherwise (T4). Two participants were undecided, and only one participant agreed. Participants were divided on whether the order of IDAPO’s steps was sensible (T5). Also, participants were equally
divided on whether storing information per step in a spreadsheet was useful (T6).

The researcher also gathered results from a few open questions. Some suggestions were:

- A spreadsheet was not considered suitable to record intermediate information;
- The process should be made less sequential;
- After identifying type and domain, always read documentation to learn about components and used technologies.

The main challenges encountered by the treatment group were:

- To find the right documentation and information;
- Lack of time to read source code, and also to identify patterns;
- Unfamiliarity with JBoss.

9.7.9.2 Results of control group

Table 9-12 lists the questions and the scores for the control group. The results for C1 show that most participants either disagreed or were undecided on whether they knew what steps to take to identify architecture patterns. Only one participant indicated he knew what approach to take. This confirms our assertion that there is a need to provide some guidance in this task. The second question (C2) was to find out whether participants found sufficient information to identify patterns. Again, most participants indicated disagreement or neutrality. This suggests that, in general, there is a need to identify useful sources of information.

<table>
<thead>
<tr>
<th>ID</th>
<th>Question</th>
<th>TD</th>
<th>D</th>
<th>N</th>
<th>A</th>
<th>TA</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>I knew what steps to take to perform the assignment.</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>C2</td>
<td>I found sufficient information to identify patterns.</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

The researcher also asked the participants for suggestions for improvement as well as challenges encountered. Some suggestions included:
Use of a debugger to trace the execution to find relations among components;
Search for images of the architecture that may lead to useful sources.

The main challenges encountered by the control group were:

- Unfamiliarity with JBoss; getting to know the system;
- Studying the source code is like finding a needle in a haystack;
- Finding the right information; inconsistent documentation; pattern naming is inconsistent (e.g., a ‘proxy’ component implementing the ‘dispatcher’ pattern)

9.7.10 Threats to validity

The researcher is aware of a few threats to validity. These threats fall in the four standard classes of threats as discussed by Wohlin et al. (2000): conclusion validity, construct validity, internal validity and external validity. These are discussed next.

9.7.10.1 Conclusion validity

The number of subjects is a threat to conclusion validity. Fourteen subjects were willing to participate in our experiment, which were divided into two groups (control, treatment) of seven. However, the researcher did not intend to make conclusive statements based on this single experiment only. Rather, the results should be considered exploratory and help to gain insights in the usefulness of IDAPO.

9.7.10.2 Construct validity

There are a few limitations to construct validity. Firstly, the researcher limited the total time for identifying patterns to two hours. This limitation has a direct effect on the amount of work that can be done, and therefore on the number of patterns that can be identified. Participants of the treatment group may have had to spend relatively much time on understanding the steps of IDAPO. However, the researcher chose to limit the time duration in order to be able to recruit a sufficient number of subjects; as the time duration of an experiment increases, fewer participants will be willing to participate.
9.7.10.3 Internal validity

There are a number of threats to internal validity, which is concerned with the degree to which a change of the dependent variable can be ascribed to a change of the independent variable. The first is *instrumentation*: the process description and diagram of IDAPO may not have been easy to understand by the treatment group. Though the researcher explained the process to the treatment group, participants may not have fully understood the steps to take. Another instrumentation threat is our marking scheme for assessing “correct” patterns. This instrument (discussed in Sub-section 9.7.6) was used to perform calculations of precision and recall. Some of the conclusions depend on the extent to which the researcher correctly confirmed the patterns. The researcher derived this trusted subset of patterns based on three independent sources. However, the three reports do not fully agree on the patterns. In order to decide which patterns to include in the trusted subset, the researcher has (a) studied the description of how the patterns were implemented, thereby assessing the credibility of the description and the pattern’s usage, and (b) attempted to find additional information through web searches in order to be able to confirm them. It is noteworthy that the patterns listed by the three sources that could not be confirmed (and therefore were not included in the trusted subset) were not identified by either group. Besides this, JBoss may contain patterns that have not been listed by any of the three reports, which means that these are not included in the trusted subset. The second threat is that of *selection*: the control and treatment groups may not be as equivalent as the researcher intended in terms of work experience and knowledge of related topics (see Table 9-6). Furthermore, the average age in the control group was 2.5 years higher (26½) than the average age in the treatment group (almost 24); this difference suggests that the control group has a few more years of experience in the field of software engineering. This could have negatively biased the results of the treatment group, which strengthens the decision to reject hypothesis $H_{06}$. 

9.7.10.4 External validity

Threats to external validity are those that may limit the applicability of the results to industry practices. Sjøberg et al. (2002) have advocated designing realistic experiments in software engineering. Citing Harrison (2000) they list three facets of realism:

- Realistic tasks: size, complexity and duration of the task.
- Realistic subjects: who performs the task (subjects of the experiment)?
- Realistic environment: the tasks may be carried out in an unrealistic manner, independent from who performs the task.

In other words, Sjøberg et al. argue that: “The experiments would be more realistic if they are run on real tasks on real systems with professionals using their usual development technology in their usual working environment”, but also point out, “keeping control is a challenge when the realism is increased”. The dilemma with designing a research method is that one is ideally seeking to simultaneously all three facets of realism. In other words, researchers are always faced with trade-offs. Here, the researcher discusses the strategy by which this fundamental dilemma was considered.

Firstly, regarding the realism of the task at hand, identifying architectural patterns in OSS products is not a well-known, common practice in industry. The researcher is not aware of any reports that describe or advocate this practice. Therefore, one could argue that the task is not “realistic”. However, the task is rather complex, and may take a significant amount of time, thereby making it a realistic challenge. Furthermore, another potential threat to external validity is that a treatment is applied on a ‘toy’ problem. However, the researcher selected the JBoss application server for this experiment, which is an industrial-strength software product. Therefore, the researcher argues that this has added to the realism of the task in the experiment.

The second facet of realism mentioned by Sjøberg et al. (2002) is the use of realistic subjects. The research presented in this chapter has been conducted with master’s students. The use of students as subjects is
an important factor that deserves attention, and has been discussed by a number of authors (e.g., (Carver et al., 2010; Höst et al., 2000; Svahnberg et al., 2008; Berander, 2004)). Firstly, it is important to understand the target group of users of IDAPO. The task of identifying patterns could be performed by a practitioner at any level, from novice to experienced professionals. However, the researcher argues that the target group of users of IDAPO is primarily novice professionals. Experienced software developers and architects can be expected to have developed in-depth insights and approaches to understand software products. As such, the researcher argues that experienced professionals do not need as much in-depth guidance as do novices. Novice professionals, on the other hand, by definition, do not have in-depth experience, and need more extensive guidance. In this context, it is useful to understand the role of novice developers in the evaluation and selection of OSS components. The final selection decision is most likely to be made by more senior members of a software development project. Senior developers or architects will typically have the final say in the decision of which components to use. However, the evaluation of OSS components may not be solely performed by senior members such as a software architect, as they are likely to have many other responsibilities. The researcher argues that novices may play a supporting role by acquiring relevant information (such as architectural patterns). As mentioned above, the target users of IDAPO are therefore novice professionals rather than experienced and senior staff.

The researcher decided to use students as a proxy for professionals in the empirical studies reported in this Chapter. Both Tichy (Tichy, 2000) and Carver et al. (2010) have discussed situations in which they consider the use of students to be acceptable. Firstly, the use of students is acceptable when piloting experimental studies (Tichy, 2000; Carver et al., 2010). Tichy speaks of “testing” and “debugging” an experiment. Carver et al. (2010) argue that it is useful to conduct an in vitro pilot study prior to an in vivo study. Therefore, prior to conducting an in vivo study to
evaluate the usefulness of IDAPO, the researcher conducted the in vitro study with students reported in this section.

Secondly, student subjects can be used to “establish a trend” (Tichy, 2000). If an experiment with students does not indicate any improvement, it is unlikely “that a noticeable effect will magically appear among professionals” (Tichy, 2000). Carver et al. (2010) refer to this as “obtaining preliminary evidence in favour of or against a research hypothesis”.

Thirdly, Carver et al. (2010) argue that student subjects can be used to test the feasibility of a technology. Before an organisation is willing to invest time in a study involving its members of staff as professional subjects, it will want to see some evidence that the technology (or process, for that matter) is feasible.

A number of authors have addressed how well students perform as proxies for professionals. Tichy (2000) compares the situation to experiments with psychology students as proxies for the general population, and argues that computer science students are “much closer to the world of software professionals than psychology students are to the general population”. Also, he argues that, “CS graduate students are technically more up to date than the ‘average’ software developer who may not even have a degree in CS.” Kitchenham et al. (2002) suggest that the use of students is sometimes criticised for making the results of formal experiments less practical. Kitchenham et al. argue that this is not a major issue as long as one is interested in evaluating the use of a technique by novice or nonexpert software engineers, which has been the case in this research.

Runeson (2003) investigated the differences between freshmen and graduate students, and found that on average, graduate students spent significantly less time than freshmen students, and therefore concluded that there is a difference between the two groups of people. On the other hand, Remus (1989) compared undergraduate students and managers in making decisions, and found that the undergraduates made more costly
decisions and used less effective decision heuristics. These reports suggest that graduate/master’s students are better proxies than undergraduate students.

Sjøberg et al. (2002) raise the point that the dichotomy between “students” and “professionals” is over-simplified. In some of their experiments, 40% of students were either working part-time during their studies, or had previously been working in industry. Many of the students involved in the research reported in this chapter also had part-time software development jobs.

The third facet of realism mentioned by Sjøberg et al. is a realistic environment; that is, offering an environment “that resembles an industrial development environment” (2002). As already mentioned, identifying architectural patterns is not a common industry practice. The researcher conducted the experiment in a computer lab, and was available throughout the sessions for questions. A colleague researcher was available to assist the researcher, but this proved not necessary.

9.8 FUTURE WORK

The experiment study was conducted as a third iteration of the Improvement Paradigm. The results of the experiment provide an understanding into how well IDAPO can guide the task of identifying architectural patterns in OSS products. This section assesses opportunities for improving the process, and how this could be packaged. However, this third iteration is not complete, in that this section only presents suggestions of how to improve IDAPO. As such, this section presents suggestions for future work.

The researcher believes that the process steps should become more flexible, and become part of a process framework, which can be tailored to the user’s needs. Furthermore, it would be valuable to investigate how IDAPO can support identification of architectural tactics, such as documented in (Bass et al., 2003). Tactics support the achievement of
quality attributes and can therefore provide valuable insights similar to the information conveyed by patterns.

9.9 CHAPTER SUMMARY

Knowledge of architectural patterns that are present in an OSS product can provide valuable insights to integrators, as patterns have a direct and documented effect on the component’s quality attributes, and as a result also on the QAs of a system as a whole. Identifying architectural patterns in an OSS product is a challenging task, however, and existing tools cannot automatically perform this task. To address this challenge, the researcher followed the Improvement Paradigm (Parra et al., 1997), which is an iterative approach for process improvement, to design a process for Identifying Architectural Patterns in OSS, named IDAPO. This chapter has presented the design and results of the studies that were conducted in three separate iterations. IDAPO provides a step-wise, systematic approach at gathering information from an OSS product with the aim to identify architectural patterns that are present in that OSS product. The process was evaluated in a quasi-experiment using graduate students as subjects. Though the results did not show a significant improvement, the results do support the identification of potentially present (“candidate”) patterns.

IDAPO is not intended to replace existing OSS evaluation methods, but rather to be used in complement to them. Existing evaluation methods, an overview of which was provided in Chapter 7, provide important insights into factors that are specific to OSS projects, such as the support to expect from a project’s community. IDAPO can be used to gather information about a product’s architectural patterns, which in turn may provide insights into a product’s quality attributes which aids in the evaluation and selection process of OSS products. As such, IDAPO offers support to practitioners in product development with Open Source Software.
Whereas this chapter as well as Chapters 6 to 8 focussed on Open Source Software, Chapters 10 and 11 which follow next focus on Inner Source, which refers to the adoption of OSS development practices within the confines of an organisation. In an Inner Source organisation OSS development practices are adopted for in-house software development. As such, this software can also be considered “software from the bazaar”, where the bazaar is internal to an organisation. Chapter 10 presents the results of a study to investigate what challenges may arise in an Inner Source context. Though the research question is similar to Research Question 1 (presented in Chapter 6), there is much more literature that reports on the use of OSS in product development than there is on Inner Source. Therefore, whereas Chapter 6 presented the results of a literature review, and as such provides a more extensive overview, Chapter 10 presents the results of one industrial case study.

Whereas the results of Chapter 10 provide insights in challenges (and approaches to mitigate them) that may arise in an existing Inner Source environment, Chapter 11 presents the results of an investigation of where and when Inner Source may thrive. As such, it provides guidance to an organisation in assessing its compatibility, or fit, with Inner Source. Together, Chapters 10 and 11 focus on Inner Source, which is a different flavour of “bazaar”.
Chapter 10

Challenges in Inner Source*

10.1 INTRODUCTION AND CHAPTER LAYOUT

There are various challenges related to the use of open source software; Chapter 6 presented an overview and classification of these challenges. While these challenges are not all unique to development with OSS components, OSS integrators do need to consider the nature of this type of software’s supplier, namely a “Bazaar” style community.

As outlined in Chapter 3, in the last decade or so, another Bazaar-type software development environment has emerged, namely the Bazaar in the Cathedral. That is, a number of software developing organisations have adopted OSS development practices, thereby setting up an “open source” community within the confines of an organisation. Little is known about challenges in product development within such Inner Source organisations. In order to investigate this issue, the researcher has conducted a case study at an Inner Source organisation.

The chapter proceeds as follows. Section 10.2 presents the detailed design of the case study. Section 10.3 discusses how Inner Source was implemented in the case study organisation, which is referred to as “SoftCom”. Section 10.4 presents the challenges that were identified in the case study. Section 10.5 compares the challenges from Section 10.4 to the challenges related to product development with OSS presented in Chapter 6. Section 10.6 presents a number of approaches that the case study organisation has adopted to deal with the challenges. Section 10.7

discusses the implications of the findings of this study. Limitations of the study are presented in Section 10.8. Section 10.9 summarises this chapter.

10.2 RESEARCH DESIGN

This chapter addresses one of the six general research questions defined in Chapter 5, namely RQ5:

*RQ5: What are challenges and approaches to mitigate these in Inner Source?*

This study aims at increasing insight into Inner Source adoption and challenges within an Inner source Software development setting. The researcher asserts that each implementation of inner source is tailored to a particular organisation; hence, it is imperative to first understand what OSS development practices an organisation has adopted. This is addressed in Section 10.3.

After outlining the Inner Source development practices in this case study, the researcher was interested in identifying the challenges that arise when integrating software components developed in-house through applying OSSD practices. Hence, the first specific research question of this study is (addressed in Section 10.4):

*SRQ1: What are the challenges in developing and using software that is developed as a shared asset?*

The researcher then intended to compare these results to the challenges related to using OSS components in product development, which were presented in Chapter 6. Hence, the second specific research question is (addressed in Section 10.5):

*SRQ2: What are the similarities between challenges in integrating OSS and challenges in integrating ISS?*

The researcher was also interested in identifying the approaches that the studied organisation had adopted to address these challenges. Therefore, the third specific research question is (addressed in Section 10.6):
SRQ3: What are the approaches used to address challenges related to integrating a shared asset?

10.2.1 Research method

As outlined in Chapter 5, the researcher chose to conduct a case study to address the main research question. The unit of analysis in this case study is an organisation that has adopted an Inner Source approach as a whole. The study was conducted at one of the locations of a large (globally distributed) organisation, which has been involved in several OSS related projects and has adopted a project-based Inner Source program. The organisation was approached through a professional contact of the researcher’s supervisors (referred to as “contact” from hereon). The organisation develops both hardware and software for safety-critical systems. For confidentiality reasons, the studied organisation’s specific domain cannot be reported. In the remainder of this chapter, the organisation is referred to by the name “SoftCom”.

10.2.2 Data collection

The researcher collected data through eleven in-depth face-to-face interviews. Table 10-1 lists the participants, their division in SoftCom, and their work experience in years. Participants are referred to by identifiers P1 to P11 in order to protect their privacy. Participants had various positions in different divisions within SoftCom, such as business division manager, product manager, technology officer, software architect, software designer and product coordinator, providing the researcher with a rounded perspective from different points of view. A number of them were members of the core team that is responsible for developing the shared asset (see Chapter 3). Most managers also had prior technical experience as a software architect. All participants had extensive experience and knowledge about SoftCom as they had worked there from 10 to more than 25 years.
Table 10-1. Participants in the case study at SoftCom.

<table>
<thead>
<tr>
<th>Participant</th>
<th>Division</th>
<th>Experience (years)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>Division A</td>
<td>20</td>
</tr>
<tr>
<td>P2</td>
<td>Core team</td>
<td>10</td>
</tr>
<tr>
<td>P3</td>
<td>Division B</td>
<td>15</td>
</tr>
<tr>
<td>P4</td>
<td>Core team</td>
<td>17</td>
</tr>
<tr>
<td>P5</td>
<td>Technology office</td>
<td>26</td>
</tr>
<tr>
<td>P6</td>
<td>Core team</td>
<td>25</td>
</tr>
<tr>
<td>P7</td>
<td>Division C</td>
<td>10</td>
</tr>
<tr>
<td>P8</td>
<td>Core team</td>
<td>10</td>
</tr>
<tr>
<td>P9</td>
<td>Technology office</td>
<td>25</td>
</tr>
<tr>
<td>P10</td>
<td>Core team</td>
<td>13</td>
</tr>
<tr>
<td>P11</td>
<td>Division D</td>
<td>12</td>
</tr>
</tbody>
</table>

Prior to conducting the case study, the researcher developed an interview guide (Taylor and Bogdan, 1984), which can be found in Appendix C. The researcher chose to conduct semi-structured interviews, as these are expected to give a researcher the flexibility to go deeper into unforeseen types of information that can emerge during interviews (Seaman, 1999). All interviews were conducted at SoftCom’s location. The contact at SoftCom made local arrangements and scheduled the interviews. After receiving the contact information of all scheduled participants, the researcher sent them an introductory letter in which the aim and procedure of the research was outlined. All interviews lasted between 40 and 60 minutes, and were digitally recorded with the participants’ consent. The recordings were transcribed verbatim, in order to record as many details as possible. This resulted in approximately 150 (A4 size) pages of text.

Finally, all recordings were played back once more and crosschecked with the transcriptions in order to make sure that no information was lost during the transcription.

10.2.3 Data analysis

Data analysis is an iterative process, in particular when a researcher is confronted with a large amount of data (in this case 150 pages of transcripts). Though the research questions are clearly defined, in order to be able to manage the large amount of collected, the researcher decided it
was important to reconstruct the “story line” for each participant, and identify common themes and topics in order to be able to compare these topics. Since different participants sometimes used different descriptions for their experiences and insights, it is important to identify these common themes. This is a form of triangulation (across data sources, namely the participants of this study), which is a common procedure to establish validity in qualitative studies (Creswell and Miller, 2000).

The researcher analysed the data as follows. All interview transcripts were thoroughly read, and phrases of interest were coded with labels to reflect the topic of that phrase, following the approach described by Seaman in (Seaman, 1999). In order to ensure reliability of the findings, the researcher applied another form of triangulation, namely among different investigators (Creswell and Miller, 2000); the researcher and one of his supervisors have discussed the findings in several face-to-face meetings.

Using specialised software for qualitative data analysis (NVivo), the researcher constructed a small set of preformed labels referring to topics that were expected to arise from the data, and which were also of interest to the researcher. During data analysis, this set of labels evolved; labels were merged, added and deleted. After the initial coding, the researcher looked at groups of coded phrases and merged them into categories. This structuring of the data helped the researcher to understand and manage the large amount of information. Per category, the labelled text was exported to a Microsoft Word document, thereby grouping all related paragraphs on a particular topic in one document. This allowed the researcher to further read and analyse the data per topic.

After the researcher had acquired an initial overview and understanding of the data, he created memos in the form of visualisations of the transcripts. These were simple box-and-line diagrams; part of such a memo is shown in Figure 10-1.
Boxes represent the main topics, whereas each box may have a number of “attributes”, or sub-topics, which are short phrases connected by lines. Boxes can also be related to other boxes. The researcher created such visualising memos for each interview, which were “maps” of the transcripts’ contents, and could quickly communicate the contents of the interviews to the researcher’s supervisors.

After identifying the main topics of each interview and recognising common themes among the different interviews, the researcher re-read the coded transcripts, to identify the challenges and approaches that participants had reported. In some cases it was necessary to refer back to the original transcripts to refresh the researcher’s mind of the context. The participants often mentioned an approach to address a certain challenge after reporting that challenge. Sometimes this was clearly indicated by key phrases, such as: “So what we do now, is [...]” or “In order to address this [...]”. The challenges and approaches were listed in a spreadsheet, together with source information to easily back trace the original phrasing in the transcripts. Similar entries were merged.

The researcher had noticed that many challenges reported by different participants were related (causing or exacerbating other challenges). In order to explore and visualise these relationships, the researcher drew more box-and-line diagrams (separate from those shown in Figure 10-1). In these diagrams, boxes represented challenges and
approaches, and lines represented relationships (such as “addresses”, “exacerbates”, etc.). These visualisations finally evolved into a complete diagram, shown in Figure 10-3. This procedure of establishing a traceable, documented justification of the analysis process (transcription of the interviews, coding, memoing) by which conclusions are reached is called an *audit trail*. This is a recommended practice to establish validity in qualitative studies (Guba, 1981).

Each challenge was identified by at least one participant, while ten (out of 13) were identified by two or more participants. The researcher did not use results from earlier interviews in interviews conducted later; this means the researcher did not let participants comment or confirm on reports from other participants. Therefore, the researcher argues it is not appropriate to perform a frequency analysis of the challenges and approaches, as this may misrepresent the truth. Each participant told his/her story (following the questions from the interview guide), highlighting his/her view on the studied topic.

10.3 THE SOFTCOM ORGANISATION

In this section, details of SoftCom that are relevant to this study are described. This section also reports the key OSS practices that have been adopted as part of the Inner Source initiative in SoftCom. This sets the context in which the findings reported in 10.4 should be interpreted as each implementation of Inner Source is tailored to the organisation’s specific context and needs (as was asserted in before).

SoftCom’s products are developed as members of a SPL. Initially, a common platform used by all products in the SPL was provided to business divisions as a binary deliverable. Besides the general motive to increase software reuse, another reason for providing a common platform was that company management had planned a series of company acquisitions, whose products were to be adopted and integrated into a common architecture. Prior experience suggested that by providing a common platform, a *turf battle* about whose technology to use in such
acquisition could be avoided. Over the last decade, SoftCom has acquired a number of companies, which have become new business divisions. The software product that a new business division brought in, would be thoroughly scrutinised to see how it would fit with the common platform, and what parts could be adopted in the platform. On the other hand, the new business division would have access to the platform that provides common functionality, and could replace parts of their software by functionality provided by the platform. This results in less code for the business division to test and maintain.

A number of years ago, SoftCom decided to adopt a project-based (see Chapter 3) Inner Source approach, in which the common platform is managed as a shared asset. In the remainder of this section outlines how Inner Source has been adopted in SoftCom. Figure 10-2 shows the conceptual model of the adopted Inner Source model. This model closely resembles an OSSD approach reflecting common mechanisms found in OSSD: the Core Team represents the OSS community’s core developers; the business division is the OSS integrator that uses the OSS product in product development, and the shared asset is the OSS product. Other parts of Figure 10-2 are discussed below.

![Conceptual model of Inner Source in the SoftCom organisation. Arrows between actors, products or processes indicate the order of reading, e.g., a Business division Participates in Collaborative development.](image-url)
As previously mentioned, Inner Source refers to leveraging OSS practices within a corporate environment (Gaughan et al., 2009). This does not mean that all OSS practices are suitable to be applied within a corporate setting. Rather, when an organisation is involved in commercial software development, it only adopts practices that can help improve process and product quality in such a way that the organisation can control the process and the product release deadlines. The level to which an organisation is ‘going open’ differs from one organisation to another. Each organisation will implement Inner Source in a different way, tailored to the constraints and needs of the organisation (Gaughan et al., 2009). This variety of practices is similar to the variety of practices in OSS projects mentioned in Chapter 2. In the remainder of this section, the OSS development practices that have been applied within SoftCom are discussed.

10.3.1 Regular releases and frequent integration

As is common in many OSS projects, SoftCom has a core team, which makes regular ‘stable’ releases of the shared asset (Robbins, 2005; Feller and Fitzgerald, 2002). A steering committee consisting of a number of architects decides what new features will be included in the new version. Business divisions can integrate these releases into their product, but they may also choose to follow development of the shared asset more closely by regularly downloading the latest version; the Inner Source model enables this option. By staying closer to the latest version of the shared asset, a business division can reduce its integration efforts, as it no longer needs to make major revisions when switching from one version to another.

10.3.2 Collaborative development

One of the key characteristics of OSS development is that anybody is free to contribute. In OSS development, this typically happens by sending a patch file that contains the changes made to the source code. The patch is then peer-reviewed by trusted contributors that have write (“commit”) access to the source code repository. Contributors that have a record of
submitting high quality patches may be granted write access. In such a case, the peer-review is effectively post-commit. In a corporate setting, contributions must be more restricted in order to control the quality of contributions, especially in business- or safety-critical systems. The organisation has adopted a mechanism called *collaborative development*, in which the core team and a business unit closely collaborate on the development of a new component, or on enriching an existing component. This mechanism helps in making sure that, on the one hand, the component will fit into the architecture of the shared asset, and on the other hand implements the required functionality, as required by the business division’s domain experts.

### 10.3.3 Local changes to the source code

Business divisions are free to make local changes to the shared asset on which they build their product. This may be a solution if a division finds out about missing functionality shortly before a product release, and the core team may be unable to make the required changes on time. This situation can be beneficial to both the business division and the core team since any such changes are “bought back” by the core team. This way, a business will no longer have to reapply (and maintain) patches whenever a new version of the shared asset is released. The core team may benefit from the domain expertise that the changes may incorporate.

### 10.3.4 Tool support

While not exclusive to OSS style development, several tools typically used in OSS projects are also used within SoftCom. Development environments are standardised in SoftCom, and managed and supported by support engineers, who are members of the core team. By standardising the development environment for all business divisions (and the core team), it is easier to ensure that a code check-in does not ‘break the build’. In order to address knowledge sharing issues, a wiki was set up through which developers and architects (both from the core team and business divisions) can share knowledge. Though most information
comes from the core team, a wiki allows anyone to contribute, which is highly encouraged by the core team. The adopted wiki implementation allows for semantic annotation of the knowledge, which allows it to be reused in different contexts. Besides a wiki, a mailing list was set up that can be used by developers to ask specific questions. Architects regularly look through these questions and answer if they can. An issue tracker is also available to report and communicate problems.

10.4 CHALLENGES IN INNER SOURCE

The researcher has identified 13 challenges in the case study (numbered S1 to S13), listed in Table 10-2. These challenges are classified using the same categories identified used in Chapter 6 (the category “documentation” was renamed “documentation & knowledge”, since documentation is a means to share knowledge). As noted before, there are no issues in the categories “product selection” and “legal and business”.

225
Table 10-2. Challenges identified in the case study and references to challenges related to the use of OSS products as identified in the literature.

<table>
<thead>
<tr>
<th>Category</th>
<th>ID</th>
<th>Challenge in Inner Source</th>
<th>Challenge in using OSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Documentation &amp; knowledge</td>
<td>S1</td>
<td>Lack of documentation.</td>
<td>Lack of, or low quality documentation.</td>
</tr>
<tr>
<td></td>
<td>S2</td>
<td>Core team that develops shared asset lack domain knowledge causing lack of attention for non-functional requirements.</td>
<td></td>
</tr>
<tr>
<td>Community, support &amp; maintenance</td>
<td>S3</td>
<td>Core team must balance spending resources over required architectural refactoring and implementing requirements as requested by business divisions.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>S4</td>
<td>Business divisions’ contributions do not fit</td>
<td>Custom changes need to be maintained, which is time-consuming and may cause problems with future versions. Community may take a different approach.</td>
</tr>
<tr>
<td></td>
<td>S5</td>
<td>Core team’s reluctance to adopt business divisions’ contributions.</td>
<td>Convincing OSS community to accept changes (modifications may be too specific); contributions can be difficult or costly. Difficulty to control the architecture if not a core member.</td>
</tr>
<tr>
<td></td>
<td>S6</td>
<td>Business divisions’ reluctance to contribute to the shared asset.</td>
<td>Contributing and investing in OSS project costs resources.</td>
</tr>
<tr>
<td></td>
<td>S7</td>
<td>Business divisions treating core team as a traditional component supplier; business division does not have influence on architecture and interfaces.</td>
<td></td>
</tr>
<tr>
<td>Integration &amp; architecture</td>
<td>S8</td>
<td>Missing interfaces causing usage of private interfaces, resulting in high integration efforts when switching to a new version.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>S9</td>
<td>Missing functionality.</td>
<td>Modifications needed to implement missing functionality or fit into architecture.</td>
</tr>
<tr>
<td></td>
<td>S10</td>
<td>Integrating acquired software into the shared asset hindered by architectural mismatch.</td>
<td>Incompatibility between components or existing systems.</td>
</tr>
<tr>
<td></td>
<td>S11</td>
<td>Component-suite model of shared asset allows for too much freedom in usage, causing many test and integration efforts.</td>
<td></td>
</tr>
<tr>
<td>Migration &amp; usage</td>
<td>S12</td>
<td>Components are not designed for other use-cases (not sufficiently generic).</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td>S13</td>
<td>Difficulty in using the shared asset, configuring is complex.</td>
<td>Complexity of configuration.</td>
</tr>
</tbody>
</table>
One of the objectives of this study was to compare the challenges related to ISS to the challenges related to OSS (as identified in Chapter 6). The challenges identified in this case study are presented first in Section 10.4, addressing SRQ1. Then a mapping and comparison of challenges related to the use of OSS products (from Chapter 6) is presented in Section 10.5, addressing SRQ2. Approaches adopted in SoftCom to address the challenges are presented in Section 10.6, addressing SRQ3.

10.4.1 Documentation & knowledge

10.4.1.1 Lack of documentation

A number of participants indicated a lack of knowledge sharing and documentation to be a challenge (S1). Though the development process followed by SoftCom prescribes that design and test packs of documentation are written, one participant stated:

“Nobody reads those test packs or even the requirements packs.”
—P10, core team.

Participants indicated a strong preference of having “How-to” knowledge, and basic design documentation that is needed to use the software in a useful way. In particular, information about interfaces, architectural patterns and tactics were considered to be useful. The lack of knowledge of how to use the software makes using the shared asset difficult, an often-heard challenge in this study. It was felt that, as an integrator, one needs to know too many details about the internals.

10.4.1.2 Lack of domain knowledge

The core team designs, develops and maintains the shared asset, which is used by the business divisions. However, a challenge that the core team deals with is that they lack specialised domain knowledge of the various business divisions’ products (S2). As a result, participants reported a lack of attention paid to the non-functional requirements:
“If a component does what it needs to do with respect to the functionality, then [the core team] thinks they’re done. Non-functional requirements in particular, in the context of using a product, is an obstacle. Performance, resource usage, those are often not considered.” —P5, technology office.

A member of a business division phrased it similarly:

“The issue is often with the non-functional requirements. It could be that the architecture chosen by the supplier performs badly with our type of data.” —P11, business division.

10.4.2 Community, support & maintenance

10.4.2.1 Balancing refactoring and requirements

The core team makes regular releases of the shared asset. As the shared asset evolves, there is a need for refactoring the architecture and making other improvements. However, since business divisions plan their releases based on a new version and require new features, the need of spending resources on these maintenance activities creates difficulties for the core team (S3). One interviewee reported this difficulty as follows:

“If push comes to shove, and the next release is scheduled, and a part of the budget is reserved for improvements, then customers say: ‘nice that you want to do that, but we need feature X or Z, otherwise we can’t deliver our product’.” —P10, core team.

10.4.2.2 Contributions don’t fit

The Inner Source model enables and encourages others within SoftCom to make contributions to the shared asset. However, it was found that until some years ago, contributions would not fit the architecture of the shared asset (S4). As one participant reported:

“People would make additional pieces of software without consultation. And when you try to incorporate that into the platform [...] it turned out to be useless.” —P2, core team.

10.4.2.3 Reluctance to accept contributions

The core team is responsible for the design and maintenance of the shared asset. The core team may be somewhat reluctant to adopt contributions of business divisions (S5), since this implies adoption of the maintenance
responsibility for the contributed software as well. One participant phrased this as follows:

“I think that if [the core team] would integrate something back into their platform, then from the other groups’ point of view they would also assume the responsibility of maintenance. And from that point on they are responsible for those parts. It’s not well defined, that if a group gives something back to [the core team], who is responsible for the maintenance of that part? Everybody thinks it would be [the core team]. And that restricts that road back.” —P5, technology office

Several factors may exacerbate this. Firstly, it may be due to the ‘not invented here’ syndrome. Secondly, contributions made by business divisions may be too specific for the business division that wrote them, rendering them unusable for other divisions.

10.4.2.4 Reluctance to contribute

Business divisions are typically not very eager to contribute to the shared asset (S6). One participant explained this situation as follows:

“Then the issue of maintenance arises: we wouldn’t mind publishing [the software], but only if the central group wants to do the maintenance.” —P1, business division.

Another reason for this reluctance appeared to be that a business division considers development of certain type of software to be the core team’s responsibility:

“When we’re doing something that’s generic, then we try to have it made by [the core team]. [...] Then we don’t want to make [that software] ourselves.” —P3, business division.

10.4.2.5 Core team as traditional supplier

A number of business divisions still treat the core team as a traditional component supplier (S7). Rather than adopting the Inner Source philosophy, these divisions have a more traditional view of software development, and do not benefit from the Inner Source model. One participant described this state of mind in these words:
“If [the shared asset doesn’t provide sufficient functionality], we send the requirements to [the core team]. They will start working on it, if all goes well (laughs). [...] They start working, developing and they’re in the basement for a while and then they come up and go: ‘Tadaa! Here you go,’ and in practice we’re not really ready for integration, so we thank them and tell them we’ll come back to them. And after a few months you use it, and then all sorts of integration issues arise. And the supplier is already in maintenance mode, and is making new things for other customers, and they don’t really have the resources to fix those problems. That’s really the biggest issue we have in practice.” —P11, business division

10.4.3 Integration and architecture

10.4.3.1 Changing interfaces

One issue while integrating the shared asset in a product is changing interfaces (S8). Interfaces of components in the shared asset are not well specified and documented, or may be private. One participant said:

“In the past there was this whole range of private interfaces that you had to use otherwise you wouldn’t get it to work.” —P3, business division.

The study revealed that other business divisions were also experiencing this challenge as another interviewee reported:

“[…] So we just use [these private interfaces]. […] Those private things can change and that will happen, and then everything collapses.” —P11, business division.

10.4.3.2 Missing functionality

A common challenge reported is that functionality is missing in newly delivered components (S9). One participant explained:

“When we get the component, it’s never been integrated, and when we do that, you find that things are missing, and without that we can’t deliver. A car with three wheels is no good…” —P11, business division.

10.4.3.3 Architectural mismatch

As new companies are acquired and integrated into the organisation as business divisions, the core team will investigate which parts of the acquired software can be adopted in the shared asset. Incorporating
software from new divisions may be troublesome (S10), as one participant described:

“We’re building a big box of Lego bricks, and they all have the same interface; Lego on Lego fits perfectly. But our stakeholders have an architecture based on Meccano, or something else, and then Lego won’t fit, and then you need to write connectors, we call that glue code. And as it turns out, the problems are always in the glue code.” —P6, core team.

10.4.3.4 Boundless reuse

Initially, the shared asset was organised as a “component suite”, a collection of components. Divisions could take whatever components they needed. However, some (acquired) divisions’ systems had only been partly adapted to the shared asset’s architecture to allow them to reuse certain components, since the component suite model allows for a ‘take what you need’ approach. This resulted in significant integration and test efforts that the organisation had hoped they could reduce through software reuse, one of the reasons to set up the shared asset in the first place. Furthermore, if the architectures of the application and the shared asset are not well aligned, there may be a need to write connectors (or ‘glue code’). Glue code may introduce problems, as described above.

10.4.3.5 Use-case mismatch

The shared asset contains functionality that is common to all business divisions. New components are being added over time, as new requirements emerge, and new business divisions are acquired by SoftCom. A challenge is to make components generically suitable to all business divisions. A common problem is that components have a use-case mismatch (S12). As one member of the core team explained:

“People complain about the maturity of the component. We build them a first time, and they’re used by customers X and Y in certain products. [...] After a year there’s another customer that also wants to use it, but in a slightly different way. Sometimes they think they need to use it differently while that’s not the case. They will consider the component as immature, because it doesn’t do exactly what they want, or it wasn’t tested in that particular use case.” —P10, core team.
A member of a business division phrased it very similarly:

“\text{The nature [of integration problems] is usually a slightly different use-case of the component than what [the core team] had tested it for.}” —P11, business division.

10.4.4 Migration and usage

Several participants indicated that it is difficult to use the shared asset (S13). Since the shared asset is the platform for a software product line, it must provide functionality that is usable by different business divisions in different specialty domains of a common industrial domain. The core team is well aware of this issue, which was reported by one of the interviewees in the following words:

“\text{We created a platform that is used by all business divisions, and because you need to keep everybody happy, it has many configuration options. [...] And I think that’s one of the reasons that it’s difficult to configure it correctly.}” —P8, core team

Business divisions have difficulty understanding how to use the shared asset, and how it relates to their product:

“\text{How do we ‘click in’ our specific application? That’s an interface issue, and well, how to pass in the data.}” —P11, business division.

Interestingly, after seeing the situation from another business division’s perspective, people would increase their insights. A member of the core team described:

“\text{And what we saw was that [at first] many people didn’t understand the abstractions and why we needed them. [...] And later people [after they had moved to a different department] would come to us and say: ‘now I understand why it was designed like this’.}” —P6, core team.

10.5 COMPARISON OF CHALLENGES WITH OSS AND ISS

The researcher observed that ten out of 13 challenges identified in the case study are similar to the challenges when using OSS products, as identified in the literature (see Chapter 6).
Not all challenges listed in Chapter 6 (see Table 6-7) are relevant to project-based Inner Source, or Inner Source at all. Chapter 3 has discussed a number of typical differences between the infrastructure-based and project-based Inner Source models. Table 10-3 provides an overview of the relevance of the different categories to OSS, infrastructure-based Inner Source and project-based Inner Source. The last row indicates the total number of challenges that are relevant to the various cases. Challenges in the category “Product Selection” do not apply to project-based Inner Source, since there is usually only a single shared asset that is developed. Selection is not an issue, since the use of the shared asset is strategic and planned. In infrastructure-based Inner Source, where departments or individuals make software freely available on an internal repository, these challenges could occur.

Table 10-3. Overview of relevance of challenges to OSS, infrastructure-based Inner Source and project-based Inner Source.

<table>
<thead>
<tr>
<th>Category (number of challenges)</th>
<th>Relevant to OSS</th>
<th>Relevant to infrastructure-based Inner Source</th>
<th>Relevant to project-based Inner Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product selection (3)</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Documentation (2)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Community, support and maintenance (6)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Integration and architecture (5)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Migration and usage (2)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Legal and business (3)</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Total number of relevant challenges</td>
<td>21</td>
<td>18</td>
<td>15</td>
</tr>
</tbody>
</table>

Challenges in the category “Legal & Business” are not relevant to Inner Source, since the software is closed source (which rules out any OSS license-related issues) and Intellectual Property Rights (IPR) and business-related concerns would not occur.

Figure 10-3 shows a mapping of the challenges identified in this case study (middle layer) to the challenges identified in the literature (top layer) indicated; the figure shows only the 10 challenges (out of 15 relevant to project-based Inner Source, see Table 10-3) identified in the literature that have also been identified in the case study (listed in Table
The bottom layer in the figure shows approaches that the studied organisation has taken to address some of the challenges.

Below it is discussed how the challenges related to integrating OSS identified in the literature manifest themselves in SoftCom; this mapping is indicated by open (white) arrows between elements of the middle layer to elements of the top layer of Figure 10-3.

During analysis the researcher found that certain challenges cause or exacerbate other challenges. These “root” challenges are displayed at the bottom in the middle layer whereas the challenges they cause (or exacerbate) are shown at the top of the middle layer. Pointy arrows between elements in the middle layer express a “cause” relationship, e.g., a lack of documentation may cause (or exacerbate) that contributions do not fit. Note that a challenge could be neither a “root” challenge nor caused by another (e.g., challenge S3). Furthermore, it cannot be claimed that the identified “root” challenges are the only sources that cause certain other challenges; it is possible that other factors are at play that have not been identified in this study.

Closed (black) arrows between elements in the bottom layer (approaches) and the middle layer indicate an “address” relationship, e.g., “providing training” addresses the challenge “difficulty in using shared asset”. There are three challenges that could not be mapped to any of the challenges identified in the literature (S2, S7, S12); these are coloured light grey. Three challenges are not addressed by any approach (S3, S5, S6); these are coloured dark grey in Figure 10-3. The researcher discusses the mapping of challenges in the remainder of this subsection. (Only those challenges that were also identified in the case study are discussed).

10.5.1 Lack of documentation (C4)

Lack of documentation is a common complaint in OSS, and in software in general. It was no surprise that in this study a lack of documentation was raised as a challenge. A lack of documentation (on how to use the
particular product) was considered to be an issue, and it also exacerbated the challenge of using the shared asset.

### 10.5.2 Dependency on community (C6)

Business divisions that base their product development on the shared asset have a dependency on the core team in a similar way to an organisation using an OSS component that becomes dependent on the community for new updates. In inner source, the core team must balance its resources spent on providing implementation of new features on the one hand, and performing architectural refactoring on the other.

### 10.5.3 Maintaining custom changes (C7)

In both OSSD and ISSD, an integrator may choose to make custom changes. However, in both cases, it is preferred to give back those changes to the community/core team, in order to prevent additional efforts needed to maintain those custom changes. (Depending on whether an OSS integrator redistributes the changed software, it is in fact required to give back those changes, as is prescribed in OSS licenses.) In both cases, the integrator may take a different approach that is incompatible with the vision of the core team, resulting in contributions that do not fit.

### 10.5.4 Getting contributions accepted (C8)

OSS integrators have experienced issues in getting contributions to the OSS products that are integrated. This issue was also experienced in SoftCom. Various business divisions may want to make changes to the shared asset, but experience obstacles in getting their contributions accepted.

### 10.5.5 Contributing costs resources (C11)

An issue reported in relation to integrating OSS products is that contributing to an OSS project costs resources. In the case of OSS, organisations may not see the benefit, or just decide that it costs too many
resources. In SoftCom, the researcher also found that business divisions may be reluctant to contribute to the shared asset.

10.5.6 Backwards compatibility concerns (C12)

As OSS evolves, new versions are released. The speed with which an OSS product evolves depends on the OSS project’s maturity and the liveliness of the community. Such changes could be product features, bug fixes and architectural changes. In SoftCom this challenge was manifested as changing interfaces of the shared asset. In development of products, business divisions need to choose for a certain version of the shared asset to use. Whenever there is a need to migrate to a newer version, interfaces may have changed, which results in additional cost to fix.

10.5.7 Modifications needed (C13)

When an organisation integrates an OSS component into a product, it may have to make modifications by implementing missing functionality, or to make it fit within the architecture of the product (Ven and Mannaert, 2008). A similar challenge was identified in the case study; a business division may find during integration that functionality is missing, or that quality requirements are not adequately addressed resulting in, for instance, poor performance.

10.5.8 System incompatibility (C14)

When integrating OSS products, each having been developed independently in a different context, incompatibility issues are bound to arise. The same challenge arises when software developed at other organisations (acquired through company acquisitions) is attempted to be integrated. Such software was developed in a certain business context, with certain requirements and restrictions. As a result, architectural mismatch (Garlan et al., 1995) may occur, which is a common challenge for architects.
10.5.9 Horizontal integration issues (C15)

When integrating software components, different integration issues may arise. Horizontal integration, as opposed to vertical integration, refers to the integration of components on the same “level”. (In vertical integration, software components would be built on top of each other, such as integrating an application on a particular runtime platform). Horizontal integration issues could be as simple as syntactic mismatch (i.e., different implementation languages). In SoftCom, horizontal integration issues were manifested as integration issues of the component suite. Initially, the shared asset was a collection of components, and since there were no bounds in the ways these components could be integrated, integrators experienced significant integration issues.

10.5.10 Configuration complexity (C17)

The complexity of configuring OSS products can become a problem when using OSS products. The researcher identified a similar challenge in SoftCom. Since the shared asset is to be used in different products, it needs to provide sufficiently abstract interfaces and use-case scenarios that fit different types of applications. This need for generic behaviour causes that the shared asset becomes very difficult to use for the integrators.

10.6 APPROACHES

SoftCom has adopted a number of approaches to address the identified challenges, as shown in the bottom layer in Figure 10-3.

A number of these approaches are OSS practices that have been adopted, described in Section 10.3. Besides these practices, SoftCom has adopted a number of other approaches to address those challenges. Other organisations that have implemented (or plan to implement) an Inner Source model, as well as OSS projects, could benefit from such approaches. These are addressed next.
Figure 10-3. Mapping of the challenges identified in the literature, the case study, and SoftCom’s approaches to address them. Challenges coloured dark grey do not have associated approaches to address them. Challenges coloured light grey were identified in the case study but were not previously identified in the literature as a challenge with using OSS.
10.6.1 Wiki and mailing lists

In order to address a lack of documentation, SoftCom has set up an internal wiki and mailing list. The wiki facilitates knowledge sharing between the core team, which supplies the shared asset, and the business divisions that integrate the shared asset. A member of the core team described:

“What we’ve been working on for the last 2-3 years is setting up a wiki around everything we do and have in [our platform]. Our developers write articles on how to use the products, and that’s really appreciated.” —P10, core team

While the wiki has proven to be an improvement in sharing knowledge, this is still a challenge. It is difficult to transfer knowledge, and there remains to be a “gap”, according to one member of a business division:

“The core team has tried to solve that by setting up a wiki, and that does help us, but it’s only one step. It was significant, because there was a huge gap, but there is still a gap.” —P11, business division

Besides a wiki, there are also mailing lists, which facilitate direct communication between customers (of the shared asset) and developers of the core team. In particular, developers can ask concrete questions regarding particular issues. A member of the core team described this as follows:

“[...] and we also set up mailing lists for concrete questions, like, ‘hey, this is my problem’. We have set up a community to answer those types of questions, and our developers and architects follow these lists, and if they recognize a question they’ll answer it, and that works quite well.” —P4, core team

10.6.2 Define API under change control

By more explicitly defining and managing a public API under change control, the interfaces of the shared asset will be more stable, which will reduce integration efforts when a business division migrates to a newly released version of the shared asset. One participant explained:
“In the past there was an extensive collection of private interfaces that you had to use to get things working, and the [business divisions] have been urging [the core team], like, guys, that drives us nuts, we need to define a public interface and put it under change control.” —P3, business division

10.6.3 Regular integration

The core team is also making demos with the shared asset, so that it is forced to play the role of integrator. As such, it is likely to encounter integration issues that can be subsequently addressed. This will help to overcome a lack of attention for both functional and non-functional requirements, since integration issues will be detected earlier and can be overcome in a more timely manner. One participant describes how his development team stays close to the latest version of the shared asset:

“What [our development team] does, much more than other teams, is to integrate with work-in-progress versions of the platform. We are very close on the latest development, close to the [core] team, and collaborate well together. Every three weeks we take their build.” —P3, business division

10.6.4 Make local changes

The Inner Source model allows business divisions to make custom, “local” changes to their copy of the shared asset, if necessary. In certain cases, it may be desirable that a business division has this option, as one participant explained:

“In the ideal case you’re sure that it works, and if you’re not, then you hope you’ll know soon enough, so you can test it and show that it doesn’t work what you had agreed. That way, we get the chance to solve and repair it. That’s how you’d like to do it in the normal case. But, sometimes that doesn’t work because a week before the release you find out that it might take two weeks to solve the problem. Well, then you don't have that time because the component happens to be developed in Bangalore [India], and this week turns out to be a celebration week. So, if you give the option to the business division to solve the problem themselves at such a late stage, in their own repository, then they’ll need to have access to our source code, and then you’ll have to go to an open source way of working.” —P6, core team
10.6.5 Collaborative development

Collaborative development is a project-based collaboration between the core team and a business division to develop new (or enhance existing) components. This is SoftCom’s approach to “open source” style development. Rather than letting anybody contribute to the source code immediately, there must be a certain level of control. In collaborative development, the business division provides the expertise to make sure that the component implements the right functionality (and with sufficient attention for quality requirements), while the core team can ensure that the component adheres to the general architecture of the shared asset. One participant explained this interaction as follows:

“What we do lately is, if we really need new functionality, we’re doing some kind of collaborative development. We send someone to [the core team], who helps with the design and the development of the requirements that we set for the component. That person that is lend out to the [core team] really has knowledge of [our field], and by doing so we basically secure that the decisions that are made are the right ones.” —P11, business division

10.6.6 Make demos

The core team that develops the shared asset does not integrate its own product. This means that they have little experience with the ease of use of the shared asset’s integration. Business divisions must find out how to best use the shared asset, and may encounter a variety of issues that were not anticipated. In order to address this, the core team now makes regular demo programs, which uses the shared asset. This way, the core team can experience first hand what potential issues may arise during integration. One participant described this as follows:

“The core team often thinks they’re done if the software just works. But precisely those extra things, such as the ease of use [of the shared asset] in a product development, receives limited attention. And what they introduced, a bit under our pressure [of the technology office], is that [the core team is] making regular demo versions to show what they’ve done.” —P5, technology office
10.6.7 Merge organisations

In order to speed up the architectural changes that are necessary to integrate newly acquired software into the shared asset, the new division was integrated with the core team on an organisational level. One participant who was closely involved in this described:

“We acquired a company that delivers an information and communication system, which was sold on a pay-per-use contract. The software was specially developed to support that business model. That was a very different architecture than we were using. In order to speed up the adaptation of our architecture, we merged the two different organisations. We said, the way that information and communication system works is fundamentally so different, but it would be good to adopt that into our platform. And you can only do that by having the development done in one organisation.” —P6, core team

10.6.8 Component assemblies

To address integration problems with the component-suite based model of the shared asset, the core team has started to offer the shared asset as sets of half-products, called “assemblies”. These are pre-constructed sets of components that are already integrated and tested, thereby reducing integration and test efforts. A business division may deconstruct such an assembly and replace a component with a different one, if necessary. One participant described this as follows:

“Take the LEGO® instruction booklet, and on page 8 there is half of the product that you need to build. That’s what we deliver... we ruined the fun for you as a kid, we just pre-assembled it but we didn't glue it together, so you can still disassemble it if you want. And the advantage is that you don’t have to assemble it anymore, which saves time, you don’t have to test it anymore. That’s more or less the parallel.” —P6, core team

10.6.9 Send delivery advocate

The core team can send a “delivery advocate” to a business division. Delivery advocate is one of a few roles (discussed in Chapter 3) identified in (Gurbani et al., 2010) and is a member of the core team that assists a business division to integrate the shared asset. By being physically present as a local helpdesk, it becomes easier to help business divisions to
integrate the shared asset. Naturally, this works better for business divisions that are located near the core team than for those located in different countries. One participant explained:

“I think that certainly we, but also the business divisions, underestimated the importance of early feedback. And with the current projects that’s going better. We send our people with the platform delivery, and they set up an integration desk at the customer, and if they start using the platform, then you’ll know quickly whether it works or not. And then you’re physically present to see whether it’s really an issue, or whether the misconfigured the product.” —P6, core team

10.6.10 Provide training

The core team can provide training. Architects from the core team explain the principles and rationale behind the architecture. Training is provided to business divisions in order to give them insights into the design of the shared asset, which will make it easier to understand how the shared asset can be used in a product. One participant of the core team explains:

“We also give training in the area of, what is [the shared asset], and how its architecture was designed; how was it constructed, and why. Usually, our lead architect gives an introduction during the morning, followed by training focused on various subparts of the [shared asset].” —P6, core team

This is quite a traditional way of transferring knowledge, and is also provided by so-called commercial OSS providers, such as Red Hat, which sells support for Linux distributions and JBoss Enterprise Middleware software.

10.7 DISCUSSION

In this section, the main findings from this research are discussed and some findings are compared to related work. Based on the findings, the researcher draws the key implications of the reported work for the research and practice of developing software-intensive systems with OSS and ISS. This discussion is focused on the challenges and the approaches used, and which challenges have not been addressed.
10.7.1 Comparison of findings to related work

This paper reports on one organisation (“SoftCom”) that has adopted a project-based Inner Source initiative. One of the main objectives of this study was to identify the key challenges related to integration of a shared asset (the Inner Source Software). While the study is the first to focus on identifying such kinds of challenges, some other studies have also reported experiences. In this section, the findings are compared to these experience reports as far as they reflect on integration of the shared asset.

10.7.1.1 Comparison of challenges

In (Gurbani et al., 2006), Gurbani et al. report on their experiences with the Inner Source initiative at Alcatel-Lucent (which they refer to as “Corporate Open Source”), as well as lessons learned from these experiences. The researcher observed a few commonalities and differences between their lessons and our results. These are discussed next.

Balancing refactoring and requirements (S3). The researcher identified a challenge of keeping a balance between refactoring of the shared asset and fulfilling requirements (Sub-section 10.4.2). Gurbani et al. reported on a similar lesson learned; they noted that: “it is essential to recognize and accommodate the tension between cultivating a general, common resource on the one hand, and the pressure to get specific releases of specific products out on time.” (Gurbani et al., 2006). The core team is responsible for delivering the shared asset (the common resource) and maintaining the conceptual and architectural integrity (which includes refactoring). In both cases, the core teams seem to be under pressure to deliver new features on the one hand, and maintain the shared asset’s quality on the other hand.

Contributions don’t fit (S4). SoftCom is a relatively large organisation with many business divisions, all of which use the shared asset. A problem that was quite prevalent until a few years ago was that business divisions would develop contributions, which did not adhere to the architectural design principles of the shared asset, resulting in a misfit.
The software may be quite useful for the business division involved, but would not be suitable for other customers as it was too specific. A similar problem also was reported by Gurbani et al. (Gurbani et al., 2006): “One of the most basic problems that many interviewees experienced was that developers were unaccustomed to thinking and designing solutions that were more general than their own product line.”

**Reluctance to adopt contributions (S5).** The researcher found that the core team (at SoftCom) was somewhat reluctant to accept contributions. The core team is responsible for the design and maintenance of the shared asset; therefore, this reluctance may be fed by a *not-invented here* feeling as well as the obligation for further maintenance of code written by others.

The interaction in the Inner Source project at Alcatel-Lucent seems to have been more open, and closer to the “open source” paradigm; Gurbani et al. (2006), report that “It is very unlikely for a developer in the [core team] to be cognizant of a feature being put into the code by another organisation.” This implies that it is easier for other developers (not members of the core team) to make changes to the shared asset directly. This is an essential difference with the situation at SoftCom, where it is explicitly not the case that non-core team members can make changes to the shared asset’s code directly. Instead, contributions are much more controlled through the collaborative development mechanism.

There are a number of possible explanations for this difference. Firstly, the Inner Source project at Alcatel-Lucent was a new product and implementing a rapidly evolving technology, whereas the shared asset at SoftCom started as a well-established component suite with well-defined interfaces and functionality. Secondly, there is a significant difference in the size of the shared assets at Alcatel-Lucent and SoftCom. The former was reported to count approximately 48 thousand lines of code (in 2005 (Gurbani et al., 2006)), whereas the latter consisted of several millions lines of code, and is therefore much more complex and serving a larger variety of business needs. Therefore, contributing to such a large shared asset is naturally more complex.
10.7.2 Challenges lead to other challenges

The researcher observes that certain challenges have led to other challenges; these are shown at the bottom in the middle layer (titled “case study: challenges”) in Figure 10-3. Each of these “root” challenges cause or exacerbate other challenges (indicated by pointy arrows between challenges in the middle layer). The researcher suggests that giving priority to these challenges while defining strategies to address them will have a positive, cascading effect. By addressing these root challenges, defined strategies may also indirectly address non-root challenges. The researcher notes that a challenge is not necessarily either a root challenge or “caused” (or exacerbated) by a root challenge; challenge S3, the tension of balancing implementing requirements and performing architectural refactoring, is neither caused by any other challenge, nor causing (or exacerbating) other challenges.

10.7.3 Unaddressed challenges

The three dark-grey coloured challenges (S3, S5, S6) in Figure 10-3 do not have associated approaches. The researcher notes that two of them are related to contributing (S5: core team’s reluctance to adopt contributions, and S6: business divisions’ reluctance to contribute). These challenges are two sides (sending and receiving) of the same medal, namely contribution, one of the core practices in OSSD. Both these challenges exacerbate the problem of missing functionality or insufficiently achieved quality requirements (S9). The researcher asserts that by improving the contribution mechanism, these two challenges can be addressed. As a result, this will also improve the level of mutual knowledge sharing within the organisation, thereby addressing the lack of domain knowledge in the core team.

By improving the contribution mechanism in Inner Source, the pressure on the core team to fulfil all requested requirements may also be decreased (challenge S3: balance requirements and refactoring), which will allow them to allocate more time to perform maintenance and architectural refactoring.
10.7.4 Open research questions

This research results have resulted in new insights but at the same time, it has identified a number of open research questions. These are discussed next.

10.7.4.1 Improving interaction and contributions

In Inner Source, business divisions have the right and means to make local changes to the software that is managed as an OSS asset, if so required. This partly addresses the problem that the core team, which manages the shared asset, lacks domain knowledge about certain requirements that a business division may have. This is especially useful if a business division is working towards a product release, and functionality turns out to be missing. Such changes should be given back to the core team, so as to take advantage of the Open Source paradigm. However, the findings suggest that this rarely happens. The core team heavily guards the shared asset’s architecture, and is reluctant to accept the maintenance responsibility of code that was not developed here. One common concern is that contributions may not respect the architectural principles of the shared asset. This challenge may also arise in OSS development: a case study reported in (Nakagawa et al., 2008) showed that after two years of development, the actual architecture differed significantly from the conceptual (designed) architecture. This phenomenon is also referred to as “architectural drift”. Linus Torvalds, creator and chief architect of the Linux kernel project, recently expressed that he was not pleased with the current state of the implementation (Modine, 2009).

One approach SoftCom is taking to address this challenge is to do collaborative development. However, the results suggest that the organisation would benefit if the contribution mechanism would be improved and better exploited. It would be valuable to improve our understanding of how business divisions can make higher quality contributions that can be more easily accepted by the core team.
10.7.4.2 Requirements versus refactoring
Since various business divisions use the shared asset, the number of requests to the core team for functionality and improvements can be quite high. Furthermore, besides the need to prioritise the business divisions’ requests, the core team also needs to maintain the soundness of the shared asset’s architecture. Therefore, there is a continuous need to balance the tension between fulfilling business divisions’ requirements on the one hand, and performing architectural refactoring on the other. The researcher asserts that it would be very valuable to gain a deeper insight into what lessons can be learned here from the OSS paradigm.

10.7.4.3 Improving knowledge sharing
One of the most recurring challenges that the researcher has identified is that business divisions found it difficult to use the shared asset. That is, business divisions have great difficulty in building an application based on the shared asset. Developers and architects have a strong need for “how-to” knowledge, how to use the component. Though this challenge has been partly addressed by the set up of a wiki and a mailing list through which knowledge can be shared that has increased the liveliness of the community. These measures may not to be sufficient. Hence, transferring knowledge effectively among different development teams remains to be a challenge.

10.8 LIMITATIONS OF THIS STUDY
The researcher is aware of a few limitations of this study that are discussed in this section. They are classified in limitations regarding construct validity, external validity and reliability. Since this case study has an exploratory nature, internal validity is not a concern, as there are no claims about causal relationships (Yin, 2003). (The researcher notes that the “causal” relationships between the “root” challenges (see Sections 10.5 and 10.7.1) that cause or exacerbate other challenges are not a matter of internal validity, but rather of reliability; the relationships were part of the findings, rather than being tested in this research. Threats to reliability are discussed below.)
10.8.1 Construct validity

Construct validity is concerned with the question whether the researcher measures what he or she intends to measure. This study is limited since the researcher gathered data from only one source (interviews). The number of interviews is limited to eleven. However, the researcher found that all participants provided him with more or less the same description of their experiences, which hinted at data saturation (Runeson and Höst, 2009). The researcher interviewed people from different divisions, each expressing their experiences and views, thereby providing the researcher with a rounded view of the topic at hand. The researcher found that results of all interviews were consistent, which increases the researcher’s confidence in the trustworthiness of the data. The consistency of the data gave the researcher confidence that real challenges have been identified that practitioners experience. All challenges can be traced back to at least one participant, and 10 out of 13 challenges were mentioned by two or more participants.

10.8.2 External validity

A commonly expressed concern of case study methodology is that no statistical generalisation can be achieved (Runeson and Höst, 2009). However, the goal of case study research is not to achieve statistical generalisation, but rather an analytical generalisation. This is of particular importance for studying a phenomenon such as Inner Source, since each case of it is tailored to the organisation in which it is implemented. Another organisation that has adopted different OSS practices is likely to encounter different challenges. This study is a first attempt to bring clarity about a relatively unexplored area. Section 10.3 presented a high-level overview of the Inner Source implementation at SoftCom, which provides context to interpret the findings presented in Section 10.4. Also, the case study was performed at an organisation that has adopted a project-based Inner Source initiative. Different challenges may occur in an infrastructure-based Inner Source initiative.
10.8.3 Reliability

Reliability is the level to which the operational aspects of the study, such as data collection and analysis procedures, are repeatable with the same results. Given that the researcher conducted the primary initial interpretation of the data, the issue of interpretive validity and trustworthiness of the analysis bears consideration. This issue was addressed by following three common procedures to establish validity in qualitative projects (Creswell and Miller, 2000), which have been briefly discussed in Sub-section 10.2.3. These are described in more detail below.

10.8.3.1 Triangulation

The first procedure is triangulation, which is validity procedure to search for convergence among multiple and different sources of information (Creswell and Miller, 2000). There are four types of triangulation: (1) across data sources (such as participants), (2) theories, (3) methods (such as data collection methods), and (4) among different investigators (Creswell and Miller, 2000). In this research, the researcher has applied triangulation across data sources, as he interviewed 11 participants and analysed and cross-compared their “story lines”. Through this procedure, the researcher found that challenges and approaches were consistently described. A second form of triangulation that was performed is among different investigators. In several face-to-face meetings, the researcher extensively discussed the study context, findings and conclusions with one of his supervisors.

10.8.3.2 Audit trail

Secondly, the researcher provides a traceable, documented justification of the process by which research conclusions were reached, thus providing an audit trail of the process, as recommended by Guba (1981). All interviews were recorded and transcribed verbatim in order to make sure that no data reduction occurred prematurely. The transcription of the interviews was done by the researcher, and was not crosschecked by others. This could potentially have resulted in information loss. However, the researcher believe this risk was minimised, as all transcripts were
compared once more to the audio recordings, in order to make sure that nothing was lost during transcription. A sample of the memoing and coding process is provided in Sub-section 10.2.3.

10.8.3.3 Member checking
The third validity procedure is member checking, whereby data and interpretations are taken back to the participants to allow them to confirm the credibility (Creswell and Miller, 2000). The researcher used a form of member checking whereby interviewees were subsequently provided with an initial draft of a paper that reported this research. Furthermore, after analysing the data, the researcher became aware of a number of deliverables of a research project that SoftCom had been involved in, that the researcher had access to. Some of the participants of the study had been involved in authoring these deliverables, which is why this is a form of member checking. These deliverables contained descriptions of a few of the challenges and approaches that had been identified and presented in Section 10.4 and therefore further confirmed the researcher’s findings.

10.9 CHAPTER SUMMARY
This chapter presented the results of a case study to investigate challenges in product development in Inner Source. The case study was conducted at a large, globally distributed organisation that has adopted a number of OSS development practices for its in-house software development. The study identified 13 challenges, 10 of which could be mapped to challenges related to OSS, which were presented in Chapter 6. Furthermore, the study also identified a number of approaches that the organisation has put in place to mitigate the identified challenges.

This chapter provides insights into challenges (and approaches to address them) that may arise in an Inner Source environment. As such, it may be of benefit to other organisations that have already adopted Inner Source, as it may become better aware of challenges by comparing the

---

25 In Summer 2010 a draft paper was written, which evolved into publication P8 (see Chapter 1), published in the Journal of Information and Software Technology. This chapter is based on that publication.
findings in, as well as adopt approaches that have been documented in this chapter. However, this does not provide any guidance to organisations that are considering adopting Inner Source. An organisation may not know how to go about to adopt Inner Source, or whether it suits the organisation at all. Therefore, the researcher argues that it is important that an organisation assesses its compatibility with OSS development practices. This is the focus of Chapter 11.
Chapter 11

A Framework for Assessing Organisations’ Fit with Inner Source

11.1 INTRODUCTION AND CHAPTER LAYOUT

Chapter 3 provided extensive background information on Inner Source, and also reviewed various sets of guidelines and frameworks that have been proposed to adopt Inner Source. However, it is not clear when and how Inner Source can thrive. Wesselius (2008) highlights that Inner Source is embedded in a company’s existing processes which are typically hierarchical and focus on top-down control, after which he poses the question: “How can a bazaar flourish inside a cathedral?” Similarly, but more specifically, Gurbani et al. (2006) conclude their study of a project-based Inner Source implementation at Alcatel-Lucent with the following question: “It is not clear, in general, how and when to initiate a project that can serve as a shared resource.” Based on this, the researcher defined the following research question that is addressed in this chapter:

RQ6: What are the key factors to consider in adopting Inner Source?

In order to address this research question, the researcher developed an assessment framework based on a review of the literature. As mentioned above, Inner Source is embedded in a company’s existing processes, and as such, each Inner Source implementation must be tailored to the context of an organisation. The framework highlights a number of key themes that need to be investigated to assess the “organisational fit” with Inner Source. An organisation may use the

---

26 A shared resource, or shared asset is the software that is managed as an internal “open source” project.
framework to assess its suitability to adopt Inner Source, and identify possible areas in which conflicts may arise; these are called “tension points”. In order to illustrate and further develop this framework, the researcher conducted an industrial case study at an organisation, which had indicated an interest in adopting OSS development practices.

This chapter is structured as follows. The chapter starts with an outline of the research design in Section 11.2. Development of the framework for assessing organisations’ fit with Inner Source is presented in Section 11.3. Section 11.4 presents the results of the framework-based assessment. Section 11.5 discusses limitations of the study, after which Section 11.6 concludes this chapter.

11.2 RESEARCH DESIGN

This research consists of two parts. Firstly, a literature review was conducted to develop the assessment framework. As Schwarz et al. (2007) point out, one of the purposes of a framework may be:

> to synthesise previous research in an *actionable* (emphasis added) way for practitioners. A framework can summarise academic literature in a meaningful way, offering guidelines and advice to practitioners.

The second part of the reported research focussed on illustrating the framework’s use in a real-world setting. The researcher conducted an in-depth industrial case study at an organisation that was considering the adoption of Inner Source, as was mentioned above. The remainder of this section discusses these two activities in more detail.

11.2.1 Literature review

In order to develop the framework for assessing organisations’ suitability for adopting Inner Source, the researcher conducted a review of the literature. Some of the results of this literature review were reported in Chapter 3. Sources of the review included all papers on the topic of Inner Source, as well as papers and books on OSS development. Papers on Inner Source were identified through various sources. Firstly, a few
papers were identified in the repository of OSS papers identified through Phase 1 of the systematic mapping study presented in Chapter 6 (See Figure 6-3). Secondly, the researcher was aware of a number of key papers in the field (e.g., (Gurbani et al., 2006; Dinkelacker et al., 2002; Wesselius, 2008)). Thirdly, the researcher had received a bibliography of a former colleague who had been active in the Inner Source research area.

The researcher thoroughly read through the various papers and books, and identified guidelines and “lessons learnt”. Guidelines are of a “normative” nature, in that authors suggest what practices or how these practices should be performed. Guidelines may be based on authors’ experiences and insights or based on the literature. Lessons learnt, on the other hand, are experiences reported by authors who have first-hand experience with Inner Source. Lessons learnt are typically specific to the context in which they were identified.

All guidelines and lessons learnt were recorded in a spreadsheet. The various guidelines were grouped if similar, and divided into categories. Details on how the framework was developed are presented in Section 11.3.

11.2.2 Case study

The researcher conducted an industrial case study at a branch of a large organisation in the Netherlands. The remainder of this chapter will use the pseudonym “NewCorp” to refer to this organisation. NewCorp produces systems that perform sequential processing of certain materials. The organisation develops both the specialised hardware and the software that controls the hardware. Sub-section 11.4.1 provides more background information about the NewCorp, though specific details about the organisation and its products cannot be reported for privacy concerns.

The organisation was contacted through a professional contact of one of the researcher’s supervisors. Previously, the contact had indicated an interest in Inner Source and Open Source Software. Based on this, contact involving the researcher was established in March 2010. The
researcher visited NewCorp’s location in the Netherlands and had an exploratory meeting with the contact (who is manager of the software development department) and another manager located elsewhere in Western Europe who attended the meeting through video conferencing. The goal of the meeting was to explore the organisation’s interests in further detail. During the meeting, it became clear that the organisation wanted to explore the option of Inner Source in more detail.

The researcher digitally recorded the meeting with the attendants’ consent. After the meeting, the researcher summarised the discussion of the meeting in a memo, which was sent to all parties involved.

Informed by the established literature that provides guidance on performing industrial case studies (Yin, 2003; Runeson and Höst, 2009; Verner et al., 2009), the researcher designed a case study. Prior to conducting the case study, the researcher developed a case study protocol document, which outlines the objectives, research questions, data collection procedures and data analysis procedures. The protocol was reviewed by the researcher’s supervisors.

The researcher planned the case study with the contact at NewCorp, which was scheduled to last approximately 5 weeks, starting early October 2010. The researcher was facilitated with a work desk in the open plan office, which allowed the researcher to sit amongst other employees. The work desk included a desktop PC connected to the organisation’s internal network, as well as a (temporary) email account in the NewCorp domain.

11.2.2.1 Data collection

The researcher collected data from three sources. Firstly, the researcher conducted interviews with 15 people, most of which were closely involved in the software development process. Details of the participants (numbered P1 to P15) are listed in Table 11-1; the table lists the participants’ position, the number of years of experience at NewCorp. The table shows that most of the participants had significant experience.
Table 11-1. Details of the interviewed members of staff at NewCorp.

<table>
<thead>
<tr>
<th>ID</th>
<th>Position</th>
<th>Experience at NewCorp</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>Software developer</td>
<td>18</td>
</tr>
<tr>
<td>P2</td>
<td>Software developer</td>
<td>8</td>
</tr>
<tr>
<td>P3</td>
<td>Software developer</td>
<td>9</td>
</tr>
<tr>
<td>P4</td>
<td>Software developer</td>
<td>6</td>
</tr>
<tr>
<td>P5</td>
<td>User interface designer</td>
<td>1</td>
</tr>
<tr>
<td>P6</td>
<td>Software developer</td>
<td>9</td>
</tr>
<tr>
<td>P7</td>
<td>Software developer</td>
<td>5</td>
</tr>
<tr>
<td>P8</td>
<td>System architect</td>
<td>8</td>
</tr>
<tr>
<td>P9</td>
<td>Team leader</td>
<td>5</td>
</tr>
<tr>
<td>P10</td>
<td>Software developer</td>
<td>5</td>
</tr>
<tr>
<td>P11</td>
<td>Software developer</td>
<td>10</td>
</tr>
<tr>
<td>P12</td>
<td>Software developer</td>
<td>12</td>
</tr>
<tr>
<td>P13</td>
<td>Software project leader</td>
<td>3</td>
</tr>
<tr>
<td>P14</td>
<td>Software developer</td>
<td>3.5</td>
</tr>
<tr>
<td>P15</td>
<td>Manager software and electronics department</td>
<td>7</td>
</tr>
</tbody>
</table>

The interview guide used for these interviews can be found in Appendix D. All but one of the interviews lasted approximately one hour; the interview with P5 lasted approximately 30 minutes. All interviews were conducted in Dutch, as the researcher is a native speaker. The researcher transcribed all interviews verbatim, which resulted in approximately 200 pages of text (A4 size).

As a second source of data, the researcher had access (through the internal network) to a variety of documentation, including design documentation, an internal wiki, project documentation, and so on. As a third source of data, the researcher attended a number of meetings, including a scrum meeting, a scrum of scrums meeting, an end-of-sprint product demonstration meeting, and a sprint-planning meeting.

11.2.2.2 Data analysis

The interviews were analysed in a similar fashion as for the case study reported in Chapter 10. For that reason, specific details are not reiterated in this chapter. Although the researcher interviewed people who work in four different projects, the unit of analysis in this case study is the organisation as a whole. The results of the case study are presented in Section 11.4.
11.3 A FRAMEWORK TO ASSESS COMPATIBILITY OF INNER SOURCE AT ORGANISATIONS

In order to assess an organisation’s fit with Inner Source, the researcher developed a framework based on existing guidance and frameworks in the literature, which were presented in Chapter 3.

Several authors have proposed sets of guidelines; most of the authors’ guidelines focus on one or more aspects of Inner Source adoption. For instance, Gurbani et al. (2006) conclude their study of Inner Source at Alcatel-Lucent with a number of “success factors” related to the technology that is managed as an Inner Source project, therefore focusing on the “software product”. Sharma et al. (2002) analysed traditional and OSS organisations based on an existing framework from the organisational literature. Neus and Scherf (2005) emphasise that adopting OSS development practices is mostly a cultural change within an organisation. Robbins (2005) focused on the adoption of OSS development tools to support the adoption of OSS development practices.

While the guidelines that have been proposed so far may be useful to organisations, none of them gives a holistic view of all the aspects that are involved (see Table 3-12). In order to overcome this, the researcher decided to find and aggregate and categorise the existing literature, which resulted in a framework with 17 elements. The framework is shown in Table 11-2. It is important to note that the guidelines that have been provided so far in the literature do not distinguish between project-based and infrastructure-based Inner Source models (see Section 3.3 for a discussion of these two models). The framework presented below focuses on project-based Inner Source, though literature from both project-based and infrastructure-based Inner Source reports has been used in the development of the framework.
Table 11-2. Framework for assessing compatibility of Inner Source at organisations.

<table>
<thead>
<tr>
<th>Category</th>
<th>ID</th>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software product</td>
<td>SP1</td>
<td>Runnable software product</td>
<td>There should be an initial set of components or product that is runnable and usable.</td>
</tr>
<tr>
<td></td>
<td>SP2</td>
<td>Needed by several stakeholders</td>
<td>Need to pool resources; contributions are useful to everyone. Support diversity of usage and encourage plurality of authorship; reusable in different contexts</td>
</tr>
<tr>
<td></td>
<td>SP3</td>
<td>Maturity state</td>
<td>Requirements and features not fully known at the outset. Need to evolve continuously.</td>
</tr>
<tr>
<td></td>
<td>SP4</td>
<td>Utility v. simplicity</td>
<td>Balance between a component’s functionality and simplicity.</td>
</tr>
<tr>
<td></td>
<td>SP5</td>
<td>Modularity of software product</td>
<td>Should be highly modular to enable parallel development of the code.</td>
</tr>
<tr>
<td>Development practices</td>
<td>DP1</td>
<td>Requirements elicitation</td>
<td>Requirements are asserted after the fact.</td>
</tr>
<tr>
<td></td>
<td>DP2</td>
<td>Implementation &amp; quality control</td>
<td>Truly independent peer review.</td>
</tr>
<tr>
<td></td>
<td>DP3</td>
<td>Release management</td>
<td>Release early, release often.</td>
</tr>
<tr>
<td></td>
<td>DP4</td>
<td>Maintenance</td>
<td>Maintenance as reinvention.</td>
</tr>
<tr>
<td>Tools &amp; Infrastructure</td>
<td>TI1</td>
<td>Development tools</td>
<td>Common tools.</td>
</tr>
<tr>
<td></td>
<td>TI2</td>
<td>Infrastructure for open access</td>
<td>Infrastructure to facilitate open access to everybody in organisation. Wiki.</td>
</tr>
<tr>
<td>Organisation &amp; Community</td>
<td>OC1</td>
<td>Work coordination</td>
<td>Task selection v. assignment.</td>
</tr>
<tr>
<td></td>
<td>OC2</td>
<td>Communication</td>
<td>Electronic and asynchronous communication.</td>
</tr>
<tr>
<td></td>
<td>OC3</td>
<td>Leadership &amp; decision making</td>
<td>Core team; reputation and meritocracy; trusted lieutenant.</td>
</tr>
<tr>
<td></td>
<td>OC4</td>
<td>Motivation &amp; incentives</td>
<td>Start small, demonstrate value; motivation; facilitate change.</td>
</tr>
<tr>
<td></td>
<td>OC5</td>
<td>Open development culture</td>
<td>Promote an open development culture.</td>
</tr>
<tr>
<td></td>
<td>OC6</td>
<td>Organisational support</td>
<td>Need to have backing from management.</td>
</tr>
</tbody>
</table>

Previously, Gurbani et al. (2006) identified six practices common across OSS projects that seem potentially incompatible with commercial software development; that is, a number of practices that may not fit well within the context of commercial software development. In their discussion section, Gurbani et al. combine some of them, resulting in four categories: (1) Requirements and software processes, (2) Work Assignment and Incentive Structure, (3) Software Architecture and (4) Tool Compatibility. These four categories can be clearly mapped to the
categories identified in the literature review, which is shown in Table 11-3. However, as can be seen in the table, the themes identified by Gurbani et al. are slightly more specific and are subsumed by the more general categories identified by the researcher. For instance, Software Architecture is only one aspect of a software product; in the proposed framework, only SP5 (modularity of software product) refers to the product’s architecture. Likewise, work assignments and incentive structure relate to coordination and motivation, which are aspects of an organisation or a community.

Table 11-3. Potentially incompatible practices identified by Gurbani et al. and how they map to the categories identified by the researcher.

<table>
<thead>
<tr>
<th>Category identified by the researcher</th>
<th>Maps to practice identified by Gurbani et al.</th>
<th>Justification for mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software Product</td>
<td>Software architecture</td>
<td>Software architecture is an aspect of the software product that is being developed as an Inner Source project.</td>
</tr>
<tr>
<td>Development practices</td>
<td>Requirements and Software Processes</td>
<td>Requirements management belongs in the development life cycle. Software processes refer to practices that are performed during development.</td>
</tr>
<tr>
<td>Tools &amp; Infrastructure</td>
<td>Tool compatibility</td>
<td>Compatibility of (software) tools is an aspect of tools (or infrastructure, which may include software tools as well).</td>
</tr>
<tr>
<td>Organisation &amp; community</td>
<td>Work Assignments and Incentive Structure</td>
<td>Work assignment is a coordination activity, which is typically handled by organisational structures and processes. Incentive structure is related to developers’ motivation; developers in traditional organisations are employed whereas developers in OSS typically have non-monetary motivations.</td>
</tr>
</tbody>
</table>

The four categories are not unrelated and independent; rather, they affect each other. This is graphically represented in Figure 11-1. These relationships (numbered a to i) are discussed in Sub-section 11.3.5 after discussions of the four categories.
The remainder of this section describes and justifies the four framework categories and their elements in more detail. The text below refers to the elements of the framework by their IDs in the table.

### 11.3.1 Software product

Several authors have suggested that the software that is managed as a shared asset exhibits certain characteristics. This sub-section discusses the characteristics of the software product that the researcher has identified in the literature. The elements in this category may help an organisation to assess whether it has suitable candidate components that may be further developed as an Inner Source project.

#### 11.3.1.1 Runnable software product

The first element of the framework is the precondition that there is an initial software product or set of components available that is runnable and usable that can be used to start up an Inner Source initiative (SP1). Raymond identified a number of necessary preconditions for bazaar style development (see Table 3-4). Raymond (2001, p.47) argues that:
It’s fairly clear that one cannot code from the ground up in bazaar style. One can test, debug and improve in bazaar style, but it would be very hard to originate a project in bazaar mode. Linus didn’t try it. I didn’t either. Your nascent developer community needs to have something runnable and testable to play with. (Emphasis by Raymond)

In other words, in order to enable developers contribute to a software product, its state of the code must be sufficient to run (S1). Wesselius (2008) reports a similar experience at Philips Healthcare: “In our experience, starting ISS development requires a set of core assets that will attract customers to the market.” Gurbani et al. (2006) suggest that, if there is no product available to set up an Inner Source program, “a research or advanced technology group is a good location to start a shared asset.”

### 11.3.1.2 Needed by several product groups

It is suggested that the software product used to initiate an Inner Source initiative is needed by several product groups or stakeholders (SP2). Based on their experiences with a project-based Inner Source project at Alcatel-Lucent, Gurbani et al. (2006) argue that the software product should be a technology that is needed by several product groups (S2), for which it makes sense to pool resources. Having different product groups use the product also helps to manage the product’s scope and reusability in different contexts (Robbins, 2005).

In the case of Alcatel-Lucent, the software product is a SIP protocol software stack, which has been integrated into products of several product groups. At Philips Healthcare, the shared asset started out as a set of components related to a much-used standard by all business units within the organisation. Over the years, this evolved into a common platform for Philips’ software product lines of medical software; various business units within Philips develop applications on top of this platform that control medical devices, such as X-ray and MRI scanners (Wesselius, 2008).
11.3.1.3 Maturity state of the software

The third element of the framework is the software product’s maturity state (SP3). Gurbani et al. (2006) suggest that the technology that is developed as an Inner Source project is:

relatively immature so that requirements and features are not fully known at the outset (and therefore there is a need to evolve continuously).

As part of his discussion of “preconditions for bazaar style development”, Raymond (2001) states that “the code may be crude, buggy, incomplete and poorly documented”, but that “it must not fail to run and convince potential co-developers that it can be evolved into something really neat in the foreseeable future”. Wesselius (2008) argues that,

It’s also important to ensure that these initial assets are ‘commodities’ with stable, well-defined specifications that can be outsourced to a central platform group.

Gurbani et al. and Raymond seem to be in agreement that the technology need not be “complete”, but that the software should be in a state that is fit to be used by others. This seems to be contradictory to Wesselius, who argues that the “asset” needs to have well-defined specifications. However, whereas Gurbani et al. and Raymond both focus on benefiting from contributions and insights from the developers’ community, Wesselius focuses on the ability to “outsource” the development of the shared asset to a central development group. The researcher argues that a balance needs to be found between benefiting from contributions (both code and requirements) from the developers community in an organisation on the one hand, and the ability to give the responsibility for the shared asset to a central group on the other hand.

11.3.1.4 Utility v. simplicity

The fourth element of the framework refers to the balance between a product’s utility and simplicity (SP4). The shared asset should be developed with a healthy balance between its utility and simplicity. Dinkelacker et al. (2002) write that, “Using the word ‘module’ somewhat
generically, it’s a challenge to strike the balance between module simplicity and utility.” Robbins (2005, p.250) writes:

Any contributor is welcome to submit a new feature to “scratch an itch” (Raymond 2001). Such willingness to add functionality can lead to feature creep and a loss of conceptual integrity. This sort of occurrence can make it harder to meet predefined deadlines, but it broadens the appeal of the product, because more potential users get their own win conditions satisfied.

A trade-off needs to be made between a component’s completeness (utility) and preventing feature creep (simplicity), which would result in a too “heavyweight” component that becomes too difficult to use.

Robbins (2005) suggests to practice reuse and reusability to manage the project scope; he outlined the tension with traditional software development, in which teams are optimising returns on their current project. As a result, the cost of providing on-going support for reusable components can be at odds with that goal. From a business perspective, Gurbani et al. (2006) argued that:

It is essential to recognize and accommodate the tension between cultivating a general, common resource on the one hand, and the pressure to get specific releases of specific products out on time.

11.3.1.5 Modularity of software architecture
The fifth element of the framework considers the modularity of a software product that is to be managed as an (internal) OSS project (SP5). Gurbani et al. (2006) argued that the initial product needs to have a sound modular architecture, which would make it feasible to merge various changes into a single development branch. This will help to allow different developers to work on different parts of the product in parallel, without causing so-called “merge conflicts” when the changes are merged into the main branch. This recommendation resonates with the common observation that OSS products exhibit a high degree of modularity.
11.3.2 Development practices

Although there is no single OSS development process, there is a recurring set of practices that are commonly associated with OSS development. Chapter 2 discussed common OSS development practices in more detail. This sub-section focuses on software development practices only, such as requirements management, coding, testing and release management. Other (non-development) practices are related to other categories; for instance, task selection and communication are coordination practices, and are therefore discussed as part of the Organisation & Community category. The remainder of this sub-section discusses some common development practices in OSS development. Assessing an organisation’s current practices based on the elements in this category will help the organisation to understand to what extent it is compatible with the “bazaar style” of development.

11.3.2.1 Requirements elicitation

The first element regarding the development practices is the requirements elicitation process (DP1). Bazaar style development differs greatly from commercial software development approaches in how a product’s requirements are gathered (Scacchi, 2002). Whereas in commercial software development there are typically processes and procedures in place that prescribe the way in which requirements are gathered, stored and managed, this is not true for OSS development. Scacchi (2004) described this as follows:

FOSS requirements take the form of threaded messages or discussions on Web sites that are available for open review, elaboration, refutation, or refinement. Requirements analysis and specification are implied activities. They routinely emerge as a by-product of community discourse about what its software should or shouldn’t do and who’ll take responsibility for contributing new or modified system functionality. The requirements appear as after-the-fact assertions in private and public email discussion threads, ad hoc software artifacts (such as source code fragments included in a message), and site content updates that continually emerge.
Through this practice, the requirements elicitation process potentially benefits from input from the wider community.

11.3.2.2 Implementation and quality control
The second aspect of the development process to consider is the implementation and quality control practices (DP2). Implementation activities in commercial development differ from typical OSS development activities, in that implementation of a feature is typically done by one or two developers at a time. The implementer develops the code, tests it, and after it has been found to implement the right functionality, the code is committed to a central source code repository. In OSS development, in contrast, the implementation activities typically involve a larger number of developers, and this involvement usually goes without any formal coordination. An OSS developer implements a feature, which is submitted to the community (typically through a mailing list or issue tracker) as a patch for review by the wider development community. Anybody with an Internet connection can access the patch to perform a peer-review and provide comments on the code. Peer-review in OSS development is “truly independent”, and may be performed by any number of interested developers, due to the open nature of OSS development.

OSS development practices are performed in parallel. That is, during development of the software, developers work on new features and debugging in parallel. Successful OSS projects that have attracted a large number of developers benefit from Linus’ Law:

Given a large enough beta-tester and co-developer base, almost every problem will be characterised quickly and the fix obvious to someone. (Raymond, 2001, p.30)

Raymond also stated this less formally as: “Given enough eyeballs, all bugs are shallow”. Fixes for defects are typically subject to the same peer-review process as patches that implement new functionality.
11.3.2.3 Release management

Release management is the third element of the development practices component of the framework (DP3). Releases can be distinguished as pre-version 1.0 (development release) and post-version 1.0 (maintenance release). Release management during the development phase in commercial software development can vary substantially, depending on the development approach that is taken. For instance, the classic waterfall approach (Royce, 1970) assumes a sequential ordering of the various stages of the development cycle (e.g., design, implementation, testing). Software is released only at the end of the development cycle after the software is considered to be ready (that is, fully implemented and tested). In more iterative development approaches, including agile approaches such as sprint-driven development, releases are done on a regular basis (e.g., after each sprint). OSS projects typically follow Raymond’s advice: “Release early. Release often” (Raymond, 2001, p.29).

11.3.2.4 Maintenance

Maintenance is the fourth element of the development practices component of the framework (DP4). Swanson (1976) identified three types of maintenance activities:

1. Adaptive changes (e.g., new functionality);
2. Corrective changes (e.g., defect fixes);
3. Perfective changes (e.g., performance improvements).

The maintenance phase is traditionally at the end of the software development life cycle, after a software product has been released to customers. Usually, a support policy is in place for commercial products which indicates for how long after the release the software is supported, before an organisation stops providing further maintenance. As older versions of the software become obsolete by newer releases, maintenance activity for these older versions may be limited to corrective changes, only addressing the most urgent bug fixes (e.g., related to security issues).

Scacchi (2004) observed that “the traditional label of software maintenance doesn’t quite fit what you see occurring in different FOSS communities”. He characterised maintenance in OSS development as a
process of “reinvention”. OSS projects slowly evolve, with many minor improvements and mutations, over many releases with short intervals.

11.3.3 Tools & Infrastructure

OSS development, being a form of global software development, greatly depends on the availability of a set of common (open source) tools to support and facilitate the software development process. By identifying the tools and infrastructure an organisation currently has in place will help to assess the organisation’s “readiness” for Inner Source as far as tools and infrastructure are concerned.

11.3.3.1 Standardise tools

A set of common development tools is considered to be important by different authors (T11). Tools commonly used by OSS projects are (besides compilers) version control systems (e.g., Subversion), issue tracking software (e.g., Trac), mailing lists and wikis (Robbins, 2005). While a number of these tools are also commonly used in commercial software development, “there is often a wider range of tools used, and it is not clear how to support open source practices in heterogeneous environments” (Gurbani et al., 2006). Similarly, Robbins highlights that traditional projects often acquire licenses for specific tools, and little standardisation across projects (Robbins, 2005, p.249). Riehle et al. (2009) reported that:

The biggest hurdle to widespread adoption of SAP Forge is its limited compliance with tools mandatory for SAP’s general software development process.

Gurbani et al. (2006) report similarly:

it is important to move toward a common set of development tools. Unlike traditional open source, the broader community of developers is constrained by the tool environments of their project work. Moving code among different version control systems in order to build a variety of products is a difficult problem, and introduces the temptation of maintaining separate forks for each project.

In a similar way, Dinkelacker et al. (2002) reported:
Migrating existing source code to a common infrastructure or user interface is a challenge, both from a technology and organisation perspective.

Organisations may choose to select and manage the required tools and infrastructure themselves, or outsource this. For instance, at Philips Healthcare, the development tools are provided and managed by CollabNet27.

11.3.3.2 Infrastructure for open access
An Inner Source organisation needs to have appropriate infrastructure to support open access to all artefacts (T12). OSS projects provide universal, immediate access to all project artefacts, allowing anyone to participate (Robbins, 2005). Open access may help to keep information widely available and up-to-date. OSS projects heavily rely on infrastructure and tools to facilitate this open access. The extent to which project artefacts are accessible to others in commercial software development settings may vary widely from organisation to organisation.

At Alcatel-Lucent, a Centre of Excellence (COE) for the shared asset was established, from which others in the organisation could download the source code and get information (Gurbani et al., 2006). This way, the COE acts as a one-stop shop for “customers” of the shared asset. Besides providing the infrastructure, it is also important to provide for sufficient support and maintenance of the infrastructure. Dinkelacker et al. (2002) argue that “IT support is absolutely crucial for maintaining uptime, running scheduled backups and recovery when necessary, and hardware maintenance as well.”

11.3.4 Organisational & community
Since Inner Source involves growing a community within the confines of an organisation, adopting organisations need to consider potential changes to the organisational structure. Furthermore, coordination processes may be in place that might be different from bazaar style

27 www.collab.net
coordination. Adopting Inner Source is not so much of a technical change but rather a social change (Dinkelacker et al., 2002).

It is important to understand the organisational structures, processes and culture that are in place. The compatibility of an organisation with Inner Source greatly depends on these organisational aspects, as they involve how the people within the organisation interact and collaborate.

11.3.4.1 Work coordination
The first element of the Organisation & Community component of the framework is work coordination (OC1). Since traditional software development organisations are typically structured in departments or projects, this affects how work is assigned to individual developers. Commercial software development is driven by project plans and release schedules. This is different from OSS development, where developers typically self-select tasks (Robbins, 2005). OSS developers, not concerned with schedules or deadlines, typically select tasks that they are interested in. The motivation to do so is often explained as “scratching an itch”. Torkar et al. (2011) identified task selection as an opportunity for commercial software development organisations to adopt a bazaar style of working.

11.3.4.2 Communication
Communication is the second element of the Organisation & Community component of the framework (OC2). A key feature of communication in OSS development is that it takes place mostly online. Communication may be synchronous or asynchronous, meaning that developers are both actively involved in a conversation, for instance through an IRC channel (synchronous), or send messages to one another through a mailing list (asynchronous). In either case, discussions may involve any number of developers or otherwise interested parties; furthermore, participants may come and go freely as they see fit. More mature OSS projects can also organise regular “meetings” in an IRC channel. For instance, the Parrot
project\textsuperscript{28} has weekly progress meetings to which all developers with a “commit bit” are invited.

This is very different from communication in commercial software development, where there is a bigger emphasis on scheduled meetings, and where face-to-face communication is much more common than in OSS projects (where face-to-face communication is minimal if not non-existent). Agile software development methods in particular value face-to-face meetings much higher than written reports and memos. Though organisations involved in Global Software Development (GSD) lack face-to-face communication as well, modern technological infrastructure (e.g., video conferencing) still allows having virtual face-to-face meetings. In any case, meetings in commercial organisations are less “bazaar-like”, in that participants are not supposed to join or leave as they please.

\textit{11.3.4.3 Leadership & decision making}

Another important aspect of organisations and communities is leadership and decision-making (OC3). In commercial software development, leadership is based on status (e.g., junior or senior software developer, manager, etc.), and decisions are made by those who have more seniority or are higher up in the hierarchy. This is very different from OSS communities, where projects are typically started by one developer who is the “benevolent dictator”. As a project grows and matures, new key developers emerge based on the concept of meritocracy (Riehle \textit{et al.}, 2009; Gacek and Arief, 2004; Neus and Scherf, 2005). Merit is earned based on contributions made to the project. Developers that are perceived to have high merit, typically have a bigger say in decisions, both regarding the software design as well as project processes. Developers that have high merit and are respected by the benevolent dictator can become “trusted lieutenants”. The Linux kernel project is perhaps the best example that follows this model closely. Most projects have a number of “core developers” who are involved daily in decision-making.

\textsuperscript{28} www.parrot.org
In Inner Source, there must be a healthy balance between leadership based on formal organisational status and merit. A few authors have suggested a focus on involving people and recognising expertise. Sharma et al. (2002) argue recognition of expertise. This could also lead to recognising “trusted lieutenants”. Gurbani et al. (2006) report that:

Some developers will naturally gravitate towards understanding sizeable portions of the code and contributing in a similar manner, often on their own time. Such developers should be recognised by making them the owner of particular subsystems or complex areas of the code (the “trusted lieutenant” phenomenon).

They further suggest the need for a special core team in project-based Inner Source, which takes responsibility for the development and maintenance of the shared asset, much like the team of core developers in OSS projects (Gurbani et al., 2010; Gurbani et al., 2006). Wesselius (2008) also reported that it is useful to have a central group to take control over the shared asset. Gurbani et al. (2010) extensively describe a number of roles and their responsibilities within a core team.

Neus and Scherf (2005) emphasise that it is important to find passionate people; to drive change you need passion, and people who understand and are excited about the change. Likewise, Dinkelacker et al. (2002) argue that the value of “champions and sponsors” should not be underestimated.

11.3.4.4 Motivation & Incentives

It is important to consider motivation of developers and product groups to get involved in an Inner Source initiative (OC4). Neus and Scherf (2005) argue that adopting an OSS development approach requires a cultural shift in the organisation, and that such a change cannot be forced but only facilitated. Wesselius (2008) argues that considering a senior management’s decision to adopt an Inner Source approach as sufficient motivation goes against the OSS development philosophy, which is characterised by voluntary participation. Neus and Scherf argue that, in order to achieve motivation and “buy-in” of staff to contribute to an Inner
Source project, it is essential to demonstrate value first, and suggest to do so by solving a concrete problem with a small scope. In a similar vein, Raymond (2001) writes:

In order to build a development community, you need to attract people, interest them in what you’re doing, and keep them happy about the amount of work they’re doing.

Wesselius (2008) discussed incentive at the level of product groups; they need to have a clear incentive to use the shared asset. Wesselius supports his argument from the perspective of business models. He reports:

To turn the current community into a bazaar, we’d have to add mechanisms that make it attractive for software groups to add software assets without involving the platform group.

In order to get started, a few authors have suggested to provide some handholding, which may eventually lead to an active community of contributors. Gurbani et al. (2010) argued that:

a certain amount of hand-holding is required to get new business divisions integrated into the Project-based [Inner Source] model to the point that they become active users, and perhaps even active contributors, of the shared asset.

Similarly, Torkar et al. (2011) have argued that defining an “entry path” for newcomers is important to allow new developers to learn, while doing productive activities. If unattended, there is a risk that new developers are placed in positions for which they are unqualified, or making their learning curve unnecessarily long. Suggested activities for newcomers include bug fixing and reducing technical debt (see Sub-section 3.5.5).

11.3.4.5 Open culture

It is important to assess the “openness” of an organisation (OC5); that is, to learn about the organisation’s culture, and if there is a strictly obeyed hierarchy. Neus and Scherf (2005) suggest to do the “Emperor’s clothes” test in order to see how open the organisation is. They describe this as follows:
We find out if there are ways in the organisation that allow a novice (e.g., an intern) to publicly call attention to the emperor’s (i.e., the expert’s) lack of clothes (i.e., to raise quality issues), or if all internal communication addressed to a larger audience has to go through some gatekeepers.

One lesson learnt reported by Dinkelacker et al. (2002) was to let go of formal structures and ownership, and highlighted the importance of “getting developers and their management chain comfortable with sharing code and development responsibility across the company”. An open development culture was also advocated by Riehle et al. (2009), who argue in favour of egalitarianism:

> Projects must exhibit a mind-set that welcomes whoever comes along to help, rather than viewing volunteers as a foreign element.

Likewise, Raymond (2001) argues that it is critical that a coordinator is able to recognise good design ideas from others. This will help to involve people.

**11.3.4.6 Management support**

A final success factor that was identified in the literature is that there is sufficient management support for an Inner Source initiative (OC6). Dinkelacker et al. (2002) describe how this was a crucial factor for one of HP’s Inner Source initiatives, which is named CDP:

> It has been critical to CDP success to have a group of executive champions. In CDP’s case we have two champions, the chief technology officer and the chief information officer. These champions provide the urgency to the organisation to start the change process.

Riehle et al. (2009) write that managers of research projects are generally supportive of the volunteer contributions, but that managers of volunteers from regular product groups are typically sceptical in the beginning. However, Riehle et al. found that they became “neutral” or even supportive once they realised the benefits of early engagement with research projects. An important aspect also is the availability of sufficient budgets for providing support.
11.3.5 Relationships between categories

The four categories are mutually related. This sub-section discusses the nature of these relationships per category, based on “outgoing” arrows in Figure 11-1 (that is, a relationship is discussed in the category from which the arrow departs).

11.3.5.1 Tools & infrastructure

Tools and infrastructure facilitate and support the development practices that developers use (a). For instance, specialised tools such as requirements management software supports practices related to requirements management (elicitation, storing, communicating, etc.). Other tools facilitate (or enable) certain practices; for instance, IRC\(^{29}\) is an effective means that enables developers that are globally dispersed (as is the case in OSS projects) to communicate and discuss in real-time. Tools and infrastructure also support coordination and planning activities, which are practices that are related to an organisation or community (b). Finally, tools and infrastructure are used to develop and manage the software product (c). Tools may have an influence on what is technically possible to produce, and can even be used to automatically generate parts of the product (e.g., generation of a parser based on a grammar specification using the YACC\(^{30}\) software program). Infrastructural software, such as version control systems (e.g., Subversion and Git) is used to manage the source code of the software product.

11.3.5.2 Development practices

Development practices are used to develop the shared asset (d). It is well known that the development process has a direct effect on the quality of a software product. This has long been recognised by researchers and practitioners, and has led to the Software Process Improvement (SPI) sub-discipline within the software engineering research arena. Which practices are, and can be used, is constrained by the organisation (e). Organisations may have legal obligations to comply with certain process

\(^{29}\) IRC: Internet Relay Chat

\(^{30}\) YACC: Yet Another Compiler Compiler, a parser generator.
standards. For instance, organisations that develop medical equipment targeting the American market have to comply with process requirements set by the Federal Drug Administration (FDA). Also, management may prescribe that certain software development methodologies (e.g., agile practices or the CMMI) are followed.

11.3.5.3 Organisation & community
People within the organisation follow the development practices (f) to develop software. Though the process and practices may be prescribed by the organisation (see relationship (e)), developers also need to “buy-in” to using these practices (Rainer et al., 2003). People within the organisation and its developers-community decide to develop a shared asset (h). This may be a strategic decision by management (such as was the case for Philips Healthcare (Wesselius, 2008)), or an initiative taken by individual developers (such as the case in Alcatel-Lucent described by Gurbani et al. (2006)). The choice of tools and infrastructure that is also made by an organisation (g). Such decisions may be made by management, but developers may also have a say in this decision.

11.3.5.4 Software product
The shared asset that is developed is used to construct products, and as such it helps to serve the organisation’s business needs (i).
11.4 ASSESSING NEWCORP’S FIT WITH INNER SOURCE

In order to illustrate and further develop the assessment framework, the researcher conducted an industrial case study at an organisation, which had indicated an interest in adopting OSS development practices.

11.4.1 The case study organisation

11.4.1.1 Overview

The visited location in the Netherlands is a research and development branch of the global NewCorp organisation with approximately 25 software developers. The visited location was founded as an independent company several decades ago, and was acquired by the global NewCorp organisation in the 1990s, which is headquartered in France. Several years ago, this organisation also acquired another company in Western Europe. Developers at the visited location now collaborate with the development teams in France and in the acquired company in Western Europe. Furthermore, some development work is outsourced to a location in East Asia.

Since a few years, the organisation has adopted a number of agile software development practices, such as the Planning Game (one of the 12 Extreme Programming (XP) practices) and Scrum. A detailed discussion of these practices is outside the scope of this chapter; instead, the reader is referred to (Abrahamsson et al., 2002).

11.4.1.2 Products

NewCorp mainly develops products in two related sub-domains, which will be referred to as “domain A” and “domain B”. Products from domain A are concerned with processing certain materials in a sequential order. Products in the related domain B are concerned with post-processing the output of products of the first sub-domain. This is shown in Figure 11-2. Customers can buy either type of products for their business needs, though it is likely that they would need both.
The visited location develops and manufactures machines of product line A. 

11.4.2 Software product

This sub-section presents the analysis of the suitability of NewCorp regarding its software products that may be developed as an Inner Source project.

NewCorp has developed a number of components that could be suitable to be developed as a shared asset, which are listed in Table 11-4. For confidentiality reasons, code names have replaced the actual component names. The first three components have been developed at the visited branch of the organisation. A fourth component (AppCommon) is a framework developed at a different branch.

Table 11-4. Components developed at NewCorp that may be candidates for Inner Source projects.

<table>
<thead>
<tr>
<th>No.</th>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>HAL ( Hardware Abstraction Layer)</td>
<td>A low-level, platform-independent abstraction layer that provides common functionality for interacting with hardware.</td>
</tr>
<tr>
<td>2</td>
<td>Detector</td>
<td>A component that detects input to the machine based on which decisions are made to process further input.</td>
</tr>
<tr>
<td>3</td>
<td>Connector</td>
<td>A component to connect to an online server to pass on diagnostic information about the machine.</td>
</tr>
<tr>
<td>4</td>
<td>Application Common</td>
<td>A high-level framework that provides common functionality for building applications.</td>
</tr>
</tbody>
</table>

Table 11-5 provides an assessment of the identified candidate products using the five elements in the “Software Product” category of the framework. The remainder of this sub-section discusses the assessment results in more detail. The discussions refer to a number of projects that are currently on-going in NewCorp; they are referred to as [ProjectX], [Project Y], and so on, for confidentiality reasons.
Table 11-5. Analysis of candidate components at NewCorp along the elements of the software product category of the analysis framework.

<table>
<thead>
<tr>
<th>Element</th>
<th>HAL</th>
<th>Detector</th>
<th>Connector</th>
<th>AppCommon</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runnable product</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Needed by several product groups</td>
<td>All future projects.</td>
<td>All but the most basic projects.</td>
<td>Most models.</td>
<td>Project B; domain B.</td>
</tr>
<tr>
<td>State of requirements and features</td>
<td>Well-defined, needs extension for new hardware; code improvements</td>
<td>Well-defined, but needs additional interfaces.</td>
<td>Finished</td>
<td>Rapidly evolving</td>
</tr>
<tr>
<td>Utility v. simplicity</td>
<td>Highly focused</td>
<td>Highly focused</td>
<td>Highly focused</td>
<td>Complex, contains much unneeded functionality for product line A.</td>
</tr>
<tr>
<td>Modular software architecture</td>
<td>Highly modular</td>
<td>Usable as module but integrated with main application of project X.</td>
<td>Highly modular</td>
<td>Foundation for application; prescribes architecture to use for application.</td>
</tr>
</tbody>
</table>

11.4.2.1 Runnable product

All four identified candidate components are runnable products that can be used by product groups. Each of these could potentially be considered to start an Inner Source project. The first three (HAL, Detector and Connector) are developed for use in products of domain A. The fourth (AppCommon) component is a framework and is developed for domain B, at a different location of the global NewCorp organisation. That branch is therefore the component’s owner, which would have to support the decision to open up the development process of the component. The source code of AppCommon was made available but was stored in a separate repository; this framework is currently used in [Project X]. Since then, the original framework has evolved significantly; if developers of [Project X] wish to upgrade to the latest version, a new merge needs to be done. This is considered to be quite an expensive and painful operation.

11.4.2.2 Needed by several product groups

Managing a component as an Inner Source project only makes sense if there are several stakeholders that need the functionality, and are
therefore potential contributors. For such components, it is sensible to pool resources, so that functionality is not implemented multiple times.

The HAL component is planned to be used in all future projects, as it provides low-level and basic functionality that is needed in virtually all products. The detector component is not used in the most elementary products, but will be used in all other products. One participant described how the HAL component would be suitable:

“It should indeed be a component that is reusable for all future projects. I can see that work for HAL, to make some kind of OSS development environment for that, so that everybody could contribute.” —P7, software developer.

The connector component is used in all products. Currently, the AppCommon framework is used in all products of domain B, and only used in one project of domain A. It is expected that the application developed using this framework will also be used in [Project Y], which is a larger version of [Project X].

11.4.2.3 State of requirements and features
The HAL component is a fairly stable and feature complete component. It communicates with various hardware components through a standardised interface. For each hardware element, a driver needs to be developed which becomes part of a driver library. The HAL component needs to be changed on occasion to add support for new hardware.

The Connector component is also quite stable and needs little further work. The Detector component, however, will need some extra functionality or interface. One participant described the need for this:

“Now, if material is fed to the machine, it will identify the codes. We also want to be able to invoke the Detector to scan a bitmap of an image, and that’s currently not possible. And the root cause of that missing functionality is, perhaps, that we as a customer of this component didn’t participate in all reviews of the component. It’s really just a functionality mismatch.” —P4, software developer
The AppCommon component is a component that is under active development, but this is under control of a different product group located at a different branch. The head of department, who initially decided to use the framework, no longer supports actively following the latest development of the framework:

“I no longer believe in a common track of AppCommon for [domain A] and [domain B]. I removed it from my 3-year plan. I tried it, it has given us enormous benefit, and I still think we wouldn’t be as far as we are today if we hadn’t used it. But it takes about half a man-year to do a merge, and assuming we do 2 merges per year, then I’d have 1 fulltime person doing merges. But if, instead, I’d let that person do maintenance specifically for things that we need, it’d be much cheaper. […] I have more faith in a common framework for [domain A] only. That’s within the scope, so I can control that.” —P15, manager software department

11.4.2.4 Utility v. simplicity

The HAL, Detector and Connector components are considered to be “stand-alone” components that can be reused by other projects, and provide specialised functionality. The AppCommon framework, on the other hand, is more of a general business application framework, which a wide variety of functionality. This has resulted in a significant complexity of the component. As one participant described:

“[There is] AppCommon, which was developed in France. Well, that’s a large piece of software, and quite heavy-weight, I think. It’s very complex. They attempted to rebuilt the .NET framework, but in a slightly different way, so whatever it contains can also be found in the AppCommon framework, but “French style” I always say. I think it contains far too much [functionality]. I always say, Keep It Simple, Stupid [KISS].”
—P2, software developer

11.4.2.5 Modular software architecture

The HAL component is a library and has well-defined interfaces. Though the component does not have a particular modular structure internally, its API is well defined and allows hardware-specific drivers to be written as additional libraries.
The Detector component was designed as a separate piece of functionality, but its implementation is currently closely embedded with the main application for [Project X]. The Detector component (consisting of both hardware for the actual interaction with materials and software to control the hardware) has also been reused in a different machine that is being developed at a different branch, whereby the system application for [Project X] is “disabled”.

“And instead of considering the detector as a component, it’s just a copy of the X application, and parts of it is replaced by parts that they need at location Y, our daughter company in [Western Europe]. They needed the detector component first, they develop more high-end machines, and we deliver a Windows CE board with the detector software, and that needs to be a complete application, and that was developed by the Detector team. So, saying that you developed a component, is subject of discussion, because they really delivered a whole application with a small part that we also use. But it’s a copy of the [Project X] software which we also use. [...] It’s not like you get an ANSI C library that you just add to the code, and it works. Without AppCommon, the Detector doesn’t work either. It’s completely based on AppCommon.” —P10, software developer

11.4.3 Development practices & processes

11.4.3.1 Requirements management

NewCorp does not have a formal requirements process that is strictly followed. The study revealed that product requirements come from a number of sources. Firstly, most requirements come from the marketing department. One participant described:

“It’s most often the marketing [department]. They ask for a machine, and deliver the specifications.” —P1, software developer

Besides marketing department, which represents the market and as such identifies opportunities for developing a new machine, there are a number of other stakeholders, as described by another participant:
“[…] so, marketing, the service people, hardware and software engineering groups, I may forget a few. They also come from the factory here, which produces the machines. Some requirements come from logistics, and also from customization, so all sorts of groups that have some stake in the machine, have their own requirements. They get together with the main architect, and they deduced the system requirements from that, and that’s sent back to the stakeholders, because there are conflicting requirements.” —P2, software developer

Finally, though requirements come from various groups, at the same time both the hardware and software engineers have sufficient domain knowledge to understand what is necessary.

“When we started, they were still working on the requirements, but we already got started. Of course, we have all those machines in the market, and fundamentally they’re all the same. [Project X] will also do these things; you already know part of the requirements.” —P2, software developer

The above quotations refer to the general requirements engineering process. The process followed for the components mentioned above varies widely. When asked how the requirements were elicited for HAL, the maintainer responded:

“I mostly did it myself. I also talk to people, of course, about how I think it should be done. It was clear that we needed a hardware abstraction layer, and for me it was clear it had to be independent of the operating system. Today it’s CMX, and tomorrow it’s something else. And I also wanted to be able to test it in a Windows environment. So, it was clear from the start there had to be an abstraction layer. Also, I started with a “tracer”, it was clear there had to be some test tool.” —P6, software developer

11.4.3.2 Implementation & quality control

Development at NewCorp takes place in sprints (Abrahamsson et al., 2002). NewCorp has experimented with different sprint-lengths, which were at the time of the study three weeks. As is common in sprint-driven development, at the start of each sprint, the development team agrees on the features to implement or to improve. After the sprint, a demonstration is given to the marketing department to show the current state of the machine.
There is quite a clear separation of work areas that the different developers within a project work on. For instance, in [Project X], two developers mainly worked on the user interface, whereas a few other developers worked on a part of the software that was referred to as “system control” (which contains the business logic and the interaction with the GUI), and another two developers were working on a part called “machine control” (which is the low-level software that interacts with the hardware). The researcher also found that to a certain extent, and in certain circumstances, some participants already work in a bazaar style fashion:

“I thought there was a bug in HAL, and that’s also some kind of extension of course. I first solved it locally, and then as some kind of review for the maintainer, sent it to him and said, I think this is what’s wrong, and this is my solution.” —P7, software developer

The maintainer of HAL described this as follows:

“Yes, usually how it works now is, for instance in [Project C], if they want to change something in HAL, then they make a diff, and send it to me for review. And I review it, and if I think it’s no problem, then I check it in.” —P6, software developer

Another participant also described how some of the development is already somewhat similar to OSS development:

“Well, in a sense we’re already doing that a bit... For the detector for instance, I’ve got 1 engineer working on it, but [Project X] often wants to do more with it than the component can do, and what happens then is that a developer of [Project X] develops those things that they want, but don’t have a real priority for us. They develop it, and then [developer X] reviews the changes. And of course there’s some discussion beforehand about how to do it. So, it hardly costs us any time, and still [Project X] has the benefit that the component can do what they need.” —P13, software project leader

Several participants also expressed that it would be important to be very cautious to make changes. One participant highlighted the importance of testing as follows:
“It is important that, if you make any changes, then it shouldn’t cause any problems in other projects. If I prepare a release, and I take the latest version, and then all of a sudden it doesn’t work anymore. Perhaps for a new machine, but not for an older one. That’s a disadvantage, and you really need to test that thoroughly.” — P1, software developer

11.4.3.3 Release management
Since development at NewCorp is sprint-driven, development releases are made after every sprint, i.e., every three weeks. Compared to more traditional (e.g., waterfall) development methods without intermediate releases, this is quite regular and similar to some OSS projects that do monthly releases (e.g., the Parrot project31). NewCorp also prepares maintenance releases for machines that are already on the market. Since the topic of “release management” in the framework focuses on development releases, maintenance releases are addressed in the next subsection.

11.4.3.4 Maintenance
As mentioned before, maintenance may occur during development or after deployment. Maintenance activities during development on the HAL, Detector and Connector components are minimal. These are typical corrective (bug fixing) or adaptive (new functionality) maintenance tasks. There is little or no time for perfective maintenance, as the creator and maintainer of the HAL component described:

“Usually, if someone needs something or asks me to make changes, then I’ll quickly fix it, maybe an hour here or there. But, for instance, I made HAL for 16 bit processors, this was 5-6 years ago, and I can see how to improve HAL for 32 bit processors. That’s a bit different; I can improve it [but] I need time to do it.” — P6, software developer

For machines that are already on the market, NewCorp prepares maintenance releases three times per year. Maintenance activities for such releases are typically of a corrective nature. That is, the focus is on

31 www.parrot.org
correcting defects, rather than adding functionality or improving existing features.

11.4.4 Tools & infrastructure

11.4.4.1 Development tools

NewCorp uses a number of tools during the development activities. Some tools are used by all projects, whereas other tools seem to be optional. All projects use the same commercial Integrated Development Environment (IDE) for implementation and compilation. Some projects use special software (Review Board) to facilitate the review process, as one participant explained:

“We use Review Board, that’s an open source tool to invite reviews, and other team members can write comments. […] That’s pre-commit, so before committing the changes, another member can check it. […] And after the commit, there is an automatic rebuild of the code.” — P7, software developer

The other member of this two-person project described how the use of Review Board is only useful if the code is sufficiently mature:

“If I make a change, then I put it into Review Board, and then [my colleague] reviews it. But, the question is when should you start using that. In the beginning I got so sick of it, because there’s so many changes on one day. It’s not production software, so to say, and then using Review Board is a huge overhead, and then I would have to wait before I could continue with the next set of changes. Now I think it’s working well, sometimes we identify some bugs. But, that might depend on the person. I heard they don’t use it in [Project X], even though they have much more code. Well, they don’t think they need it.” — P12, software developer

11.4.4.2 Infrastructure for open access

NewCorp’s internal Local Area Network (LAN) provides open access to all project artefacts to anyone who has an account on the network. Access to the source code is facilitated by the Subversion32 version control system that runs on a central server. Furthermore, the issue tracker is an online system, thereby allowing access from anywhere on the network.

32 http://subversion.apache.org/
Besides this, the organisation has a wiki installation that facilitates knowledge sharing throughout the organisation, to which any developer may contribute. The organisation furthermore uses email extensively, though not in the form of mailing lists as found in typical OSS projects. In other words, email conversations are private amongst the conversation participants.

One participant described that the tools that are currently in use at different branches of NewCorp can facilitate open access to the AppCommon component:

“What would be good to have is a common repository [for AppCommon], rather than, this is our current snapshot that you can use. It would be better to indicate, this is common for both, make it an external component for both, and then you don’t have to merge but just an update. The tools that we use are suitable for that, tools that are used in both branches.” —P14, software developer

The fact that the visited branch of NewCorp does not have access to the main repository of AppCommon, but has a snapshot copy only, results in compatibility issues, as AppCommon is under active development. In general, participants indicate it would be a good idea to work with a common archive. One participant described this as follows:

“We had a version [of AppCommon] from May or March or something. And in September they released a new version, and when we asked them how to merge it, they responded: “Well, don’t merge ours, but merge your own changes into [our version of AppCommon]”. So, the other way around; not a delta from them: this is what we changed. No, they said: “take our version as a starting point, and merge all the changes that you’ve made so far into that.” Because, they had changed so much in those 6 months, it would be too complex to merge their version. Well, then I think, perhaps it’s better to work with a common archive… now they are just 2 physically separated archives.”

—P10, software developer

11.4.5 Organisation & community

11.4.5.1 Work coordination

NewCorp follows a number of agile practices, which also guide the coordination of work that is done by the developers. For instance, as
mentioned above, development takes place in sprints. Before each sprint, a planning is made for what features will be implemented. Each workday starts with a short scrum meeting to discuss the progress and plans; these meetings are per project. At a departmental level, there is a weekly scrum-of-scrums meeting, whereby the head of the software development department meets with all projects’ scrum masters to discuss progress and planning.

**11.4.5.2 Communication**

The research & development department of the visited branch of NewCorp is divided over two floors, which may have an effect on the choice of communication. Developers which are located nearby, in eyesight, are more likely to be contacted in person rather than by electronic communication, whereas developers on a different floor may be more likely to be contacted by e-mail

The means of communication within the organisation may also depend on the nature and purpose of the conversation. For instance, one participant described a case of a proposed feature change (screen size):

“[…] At some point we had decided to go to a 12” screen, with the buttons on the display, because that’s nice since you don’t need any real buttons. But, we had to redesign the software, and in the end we stuck with a 10” screen, and then it’s handier to keep the buttons off-screen […] So I just sent an email to marketing, asking, would you agree if we keep the buttons off-screen, because then we don’t have to redesign the GUI. And the next day they had replied, ‘yes, that’s fine’.” —P13, software project leader

In other cases, it is quite common to plan face-to-face meetings to discuss design or other issues.

“For Project Z, we now have a weekly interdisciplinary meeting, with the hardware engineer, and me, and also the electronics engineer, and someone from verification. They get together to discuss the “state of the world”, and to discuss, “we’re going to add that part, does that have an impact for you?” That helps, but it’s a bit odd that we need this meeting, because you’d really expect people to just walk over, rather than waiting for the next meeting.” —P13, software project leader
11.4.5.3 Leadership & decision making

Being a commercial software development organisation, NewCorp has a formal leadership structure in place. The software development department is led by the department manager. Within the department, software for a machine is developed within a project, which is led by a project manager. Project managers report to the department manager.

Decisions about design or changes are typically initiated through so-called Change Requests (CRs) or Problem-Reports (PRs). Both CRs and PRs are reported in an issue tracker (Mashups, see Sub-section 11.3.3). A Change Control Board (CCB) meets fortnightly to discuss and plan the requested changes:

“Yes, the CCB, change control board, discusses [the issues] and then they look at the priority, an analysis is done of, what is the impact, who are the parties that use the component, so they can see whether it has any effect on the product.” —P2, software developer

The CCB acts very much like a “core team” that can be found in Alcatel-Lucent and Philips Healthcare. One participant reported that, however, in his view, the CCB is sometimes “by-passed”:

“In my view, the CCB is by-passed sometimes, in practice. Sometimes big features are implemented without a CR... and that’s mostly because... we have relatively little discipline in connecting code to PR and CR numbers.” —P4, software developer

One participant expressed his concern that the software development department is too small for having a separate component group:

“We’re too small I think. I think it’s better to have people work in a project, then you prevent they’re getting stuck in an ivory tower, and it’s better to make sure that there’s something that maintains the software base. So that the whole department shares that responsibility.” —P8, system architect

11.4.5.4 Motivation & Incentives

One of the participants indicated it would be difficult to motivate people to get actively involved in the community:
“If you’re working on something and you have a problem, then you could post it; sometimes you don’t know whom to ask. And if you would have such a channel, then you could try to explain it and ask who could help out. But well, there have to be people that look at it regularly, that’s the issue. How do you enthuse people to get them involved? […] I wouldn’t know.” —P1, software developer

Another participant highlighted the time constraints to be a major blocker:

“I don’t think it’ll happen that quickly, because you need to study the software, and it’s not like we’re turning thumbs all the time, you do have to have the time for it. It’s likely that you’re in a project, and deadlines are always overrun, so everybody is fighting against time, and then you don’t have the time to do these things on the side.” —P3, software developer

11.4.5.5 Open culture

In terms of the “Emperor’s clothes” test, suggested by Neus and Scherf (2005) (see Sub-section 11.3.4.5), the organisational culture in NewCorp can be considered quite open. This is quite different from the organisational culture at the location in France that develops the AppCommon framework. One participant described:

“Yes, [we have] very different cultures. Here, we don’t care about hierarchy at all, but there, if the boss says something, even though it’s wrong, they just do it. We’ve seen that multiple times. […] The hardware project leader there says, we’re going to do it like this. And then we say, “that’s not correct, we won’t do that”. The guys over there will [follow orders].” —P12, software developer

11.4.5.6 Management support

A number of years ago, a small, development group was set up, which had the assignment of (a) developing a common reference architecture for the two machines which were going to be developed, and (b) to develop components that would be reusable within that reference architecture. The reference architecture would therefore facilitate the reusability of the components. These components could then be reused within all machines that were based on the reference architecture. The Connector component was developed before this initiative, but was put under management of this new component group. The Detector component was developed in
this component group, and is currently maintained by one developer and a project manager. However, since the developers in this group were required to work on a new project, there are hardly any resources to sustain further development in this component group. This means that software is mostly developed within the scope, and specific for, one project. As one participant described:

“We have a detector component, which was designed to be used in different projects. We considered in advance that we would set that up as a separate component. And this is reusable in different projects, after some modifications. In general we can only develop within projects, so to say. That means that you typically develop project-specific components. We don’t have a division, except for the detector component, that develops independent libraries, independent from any project. That’s mostly a matter of budget; there is no money [for developing more generic components]. And usually that means that you won’t do that.” —P4, software developer

Though management did initially support the concept of a component group, a practical lack of resources put this initiative partly on hold.

A lack of resources is also an issue in the use of AppCommon, which is another potentially viable candidate component for Inner Source development. The visited location started using the AppCommon component, partly for political reasons to generate goodwill with headquarters and demonstrate competency. However, though every year resources are requested to provide some support for the component, any allocated resources are the first to be cut whenever there are budget cuts. The head of department (who made the decision to use the component) described it as follows:

“So, we said, it’s nice that we reuse things, but we do need some type of support. And I quickly noticed that the support from the other side [the supplier] was limited to the money that was left; let me put it that way. [...] My [colleague manager], head of software there requests 2 man year in the budget rounds to support us, for [AppCommon], and it’s the first thing that’s denied.” —P15, head of software development
11.4.6 Discussion

This section discusses the implications of the findings of the case study, and identifies tension points that organisations may have to consider in adopting Inner Source.

11.4.6.1 Software product

A number of components were identified that could serve as a shared asset within an Inner Source setting. All components are in runnable state. The HAL and Connector components may be most promising, as they are needed by most of the future projects, thereby having different stakeholders. They are also quite modular, without too many external dependencies. The Detector and AppCommon components, on the other hand may be less suitable. The Detector component is, according to the findings, too dependent on the AppCommon framework. The AppCommon framework is too complex, and is developed at a different location, and specialised for products of a different (though related) sub-domain.

11.4.6.2 Development practices

The results indicate that it is mostly the marketing department that is “itching a scratch”, although it is a business need rather than a personal need. That is, initiation of new projects comes primarily from the marketing department. New projects typically are purposefully planned to target a certain segment of the market. It is important for the developers to devote sufficient attention to the requirements gathering phase, as developers would typically not be able to assert them “after the fact”, although it was found that all products fundamentally perform the same basic operations on the materials that are processed. A key factor is that the machine exhibits a certain behaviour (user-machine interaction), which is requested by the marketing department.

Since implementation takes place in sprints, there are regular releases of the software, which conforms to the “release early, release often” paradigm. Some developers already take initiative to fix identified bugs, first locally, then sharing it with the maintainer of the (HAL)
component. In general, maintenance activities are mostly of corrective nature, as a lack of time does not allow “reinvention” which would be a form of perfective maintenance.

11.4.6.3 Tools & Infrastructure

All projects use the same commercial integrated development environment (IDE). An IDE typically manages files within a “project” or “solution”; this is a container file that keeps references to all files that are part of the implementation of the software (e.g., source code files, and resources such as XML files). Using the same IDE means that all developers can (potentially) work on different projects without the need to change development environment (e.g., compiler, editor).

Some—but not all—projects use special software (Review Board) to coordinate the review process. The software helps to keep tight control over the changes made, and can automatically notify developers of changes. Though the use of this software may make the review process more structured, it is not required to perform reviews. In OSS development, reviews are typically coordinated using basic infrastructure, such as issue trackers and email. This basic infrastructure is in place, which would allow an OSS-style approach to review.

Issues (change requests (CRs) and problem reports (PRs)) are managed through a software system called Mashups. This system facilitates the coordination of planning and assigning CRs and PRs to developers.

NewCorp’s LAN allows access to all project artefacts, which is an important prerequisite for Inner Source. Furthermore, the organisation already uses an internal wiki for knowledge recording and sharing.

Though email is used throughout the company, there is no emailing list in place. An emailing list is different from targeted email (which may potentially address a whole department or the whole organisation), in that a mailing list server provides services to manage
mailing lists\textsuperscript{33}. As such, it provides an archival function, in that it stores conversation threads in a form that can be consulted at a later date. This is different from “normal” email that is typically private to senders and receivers. Setting up a mailing list server would enable online communication (see also the next sub-section).

11.4.6.4 Organisation & Community

The key characteristic of work coordination in OSS development is self-selection of tasks; developers choose a task to “scratch an itch”, but selection of tasks is also affected by knowledge and expertise. Self-selection of tasks can still take place within a commercial software development context, but would have to be discussed and considered during the sprint planning, and addressed in the daily Scrum meetings, so that selected tasks are considered in the work “burn-down”\textsuperscript{34}.

The agile development paradigm encourages face-to-face meetings, whereas communication in OSS development typically takes place online, through mailing lists or IRC channels. Means of communication therefore seems to be one aspect in which the agile and OSS development paradigms differ. Also, whereas in OSS development decisions may be discussed openly, certain types of questions (such as the decision to change the screen size as illustrated in Sub-section 11.4.5.2) are directed specifically to Marketing department. It seems that it does not make much sense to discuss these things on an open mailing list, if such decisions are to be made not by the community, but by one particular stakeholder.

Decision-making within any organisation is typically based on the organisational hierarchy. However, opening up the decision-making process (e.g., by discussing proposed decisions on a mailing list) and thereby implicitly soliciting for input from others may help to find a broader support for certain decisions. For instance, the decision to adopt AppCommon by the head of the software development department was

\textsuperscript{33} An example of such software is Mailman, available at: http://www.list.org/

\textsuperscript{34} http://en.wikipedia.org/wiki/Burn_down_chart
initially not received with great enthusiasm by developers. Opening up the discussion may make the process more “democratic”. However, at the same time the authority of management must be respected. A balance must be found between giving the “community” a say in the decision making process on the one hand, and management’s “veto” right on the other hand to be able to make strategic decisions that go against the community’s will.

A key barrier to motivating people to voluntarily get involved in an Inner Source project is a lack of time. This is also related to the topic of work assignment and release schedules underlying the nature of software development in a commercial context.

The global NewCorp organisation consists of a number of different divisions. The visited location was founded as an independent company several decades ago. As the various locations have different origins (that is, either through acquisition or through foundation by the global NewCorp organisation), the different branches have different cultures. The culture at the visited location is quite open, whereas the culture at the location in France was reported to be much more hierarchical. Depending on the scope of the Inner Source initiative, such cultural differences need to be considered. For instance, the AppCommon component is developed and maintained at the location in France; if an Inner Source initiative is started for this component, the organisational cultures and their differences need to be considered.

Management support is essential to facilitate an Inner Source initiative. Although there is a certain willingness to support such initiatives, this is not reflected in the actual financial support.
<table>
<thead>
<tr>
<th>Category</th>
<th>Element</th>
<th>Finding from case study</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software product</td>
<td>Runnable software product</td>
<td>• Four components are viable candidates: HAL, Detector, Connector, AppCommon</td>
</tr>
<tr>
<td></td>
<td>Needed by several stakeholders</td>
<td>• HAL and Connector components seem most promising as an Inner Source project.</td>
</tr>
<tr>
<td></td>
<td>Maturity state</td>
<td>• Detector component has dependencies on AppCommon, and may be less suitable.</td>
</tr>
<tr>
<td></td>
<td>Utility v. simplicity</td>
<td>• AppCommon may be too complex; also, it is “owned” by a development branch in France. The visited location is merely a customer.</td>
</tr>
<tr>
<td></td>
<td>Modularity of software product</td>
<td></td>
</tr>
<tr>
<td>Development practices</td>
<td>Requirements elicitation</td>
<td>Fundamental requirements are already known through experience, but project-specific requirements regarding machine behaviour remains to be an important factor that affects</td>
</tr>
<tr>
<td></td>
<td>Implementation &amp; quality control</td>
<td>Sprint-driven development, planned per sprint. Some developers take initiative in fixing defects and sending it to a component’s maintainer.</td>
</tr>
<tr>
<td></td>
<td>Release management</td>
<td>Software (development) releases are performed regularly, after each sprint.</td>
</tr>
<tr>
<td></td>
<td>Maintenance</td>
<td>Little or no time for “reinvention” (perfective maintenance).</td>
</tr>
<tr>
<td>Tools &amp; Infrastructure</td>
<td>Development tools</td>
<td>Common tools are in place.</td>
</tr>
<tr>
<td></td>
<td>Infrastructure for open access</td>
<td>Infrastructure is mostly in place. A mailing server and IRC server could be offered and its use be piloted.</td>
</tr>
<tr>
<td>Organisation &amp; Community</td>
<td>Work coordination</td>
<td>Selection of tasks needs to be controlled and communicated in order to ensure correct prioritisation of work.</td>
</tr>
<tr>
<td></td>
<td>Communication</td>
<td>Communication targeted at specific stakeholders is unlikely to benefit from “open” communication.</td>
</tr>
<tr>
<td></td>
<td>Leadership &amp; decision making</td>
<td>Must find balance between authority and room for discussion</td>
</tr>
<tr>
<td></td>
<td>Motivation &amp; incentives</td>
<td>Lack of time, release schedules in commercial development</td>
</tr>
<tr>
<td></td>
<td>Open development culture</td>
<td>Visited location has open culture, but other locations are more hierarchical.</td>
</tr>
<tr>
<td></td>
<td>Management support</td>
<td>Lack of resources and budget cuts are a barrier.</td>
</tr>
</tbody>
</table>
11.5 LIMITATIONS OF THE STUDY

The researcher is aware of a few limitations of the research presented in this chapter, which are discussed next. Firstly, the framework presented in Section 11.3 was solely developed by the researcher. That is, no crosschecking by, or triangulation among, different investigators (Creswell and Miller, 2000) was performed. Prior to the case study, the framework had not been validated. This means that it was not yet clear whether the framework captures (1) the correct data, and (2) a complete view. However, the framework was based on a thorough literature review, and one purpose of the case study was to demonstrate the usefulness of the framework and identify missing elements or categories. Also, through further research, the framework may be further developed and validated. The researcher argues that the framework’s current components and elements are grounded in the literature, and as such is a useful starting point.

Finally, since this research focuses on project-based Inner Source, the framework’s focus is on this flavour of Inner Source. Though one could argue that the project-based Inner Source model is a special case of the infrastructure-based model, the framework may have to be tailored for the latter.

11.6 CHAPTER SUMMARY

This chapter has presented a framework, grounded in the literature on Open Source and Inner Source, to analyse an organisation’s fit with adopting Inner Source. The framework is an analytical tool that can assist an organisation to investigate how compatible its current state of practice is with OSS development practices. It can highlight which typical OSS development practices may be suitable to adopt, and which ones are likely to be incompatible.

To illustrate a practical use of the framework, the researcher conducted an in-depth case study at an organisation (named “NewCorp”)
that had indicated an interest in adopting Inner Source. The analysis of the state of practice at NewCorp using the elements of the framework has revealed a number of potential tension points. As such, the framework has identified the practices that support the adoption of Inner Source as well as practices that may hinder it. An organisation can use these insights to assess to which extent an organisation is compatible with Inner Source. For instance, the framework helped to assess the suitability of a number of candidate software products to be managed as an Inner Source project. Furthermore, as was discussed in Chapter 3, each organisation adopting Inner Source does so in its own way, tailoring to the specific needs and context of the organisation. The framework can help to identify which practices would be suitable to adopt and which ones would be less suitable.
12.1 INTRODUCTION AND CHAPTER LAYOUT

This chapter summarises the research presented in this thesis and presents suggestions and directions for future work. The chapter begins with a review of the research objectives and research questions in Section 12.2. Section 12.3 summarises the contributions made by the research reported in this thesis and discusses implications of these results for research and practice. The chapter ends with a number of suggestions for future work in Section 12.4.

12.2 REVIEW OF THE RESEARCH OBJECTIVES

The research objectives, as stated in Chapter 1, were defined as follows:

- To document challenges and to provide guidance in product development with OSS components.
- To identify challenges in Inner Source and to provide insights into when and how Inner Source can thrive.

Based on a thorough review of the literature presented in Chapters 2, 3 and 4, a set of six research questions were derived in Chapter 5. This section briefly reviews these research questions; Chapter 5 presents an extensive description of how these questions were derived. For the first research objective regarding “Open Source” bazaars, four research questions were defined; these are reviewed in Sub-section 12.2.1. For the second research objective regarding “Inner Source” bazaars, two research questions were defined, which are reviewed in Sub-section 12.2.2.
Research questions for objective 1: Open Source

Building software systems with OSS is an increasingly common activity. However, as of yet there has been no complete overview of the challenges that a practitioner may encounter while using OSS products. Hence, the first research question is defined as follows:

RQ1: What are the challenges associated with product development with OSS components?

One of the most common challenges that were identified is the evaluation and selection of suitable OSS components. A variety of OSS evaluation approaches have been proposed, both by researchers and practitioners. However, there is no clear overview of what approaches exist, and except for two attempts to compare a few of these approaches, they have not been compared in a systematic way. This led to Research Question 2:

RQ2: What is the state of the art of OSS evaluation and selection methods?

As Chapter 4 pointed out, software architecture has received a lot of attention within software engineering research. In CBSD, software architecture also plays a critical role. Knowledge of a component’s architecture has also been recognised to be an important aspect. However, it is not quite clear what architectural knowledge (AK) practitioners would need from OSS products. Hence, Research Question 3 was defined as follows:

RQ3: What is the importance of architectural knowledge for OSS integrators?

One of the findings of the study addressing RQ3 was that OSS integrators do, in certain cases, appreciate the availability of architectural knowledge of OSS products. One type of AK in particular is architectural patterns. Architectural patterns are an important type of AK, as patterns have a direct influence on a software product’s so-called quality attributes (QAs), such as performance and reliability. Architectural patterns are therefore a promising means to perform evaluation of software products.
Previously, patterns have been shown to be an effective means to evaluate software architectures. However, architectural patterns are often not documented, and in OSS in particular, documentation of design decisions is often lacking. One approach that practitioners could take is to try to recover patterns. There is little guidance to support practitioners in this task. Hence, Research Question 4 was defined as follows:

*RQ4: How can architectural patterns in OSS products be identified?*

RQ4 concludes the series of research questions to address the first research objective of the research presented in this thesis, namely to document challenges and provide guidance in product development with OSS components. Research Questions 5 and 6 address research objective two of this thesis, and are discussed in Sub-section 12.2.2, which follows next.

**12.2.2 Research questions for objective 2: Inner Source**

The second research objective addressed in this thesis is to identify challenges and provide guidance in Inner Source. There have been a number of case studies that have studied the Inner Source phenomenon, but little is known about what challenges may arise in such a setting. Hence, the researcher defined the following research question:

*RQ5: What are challenges and approaches to mitigate these in Inner Source?*

Though a number of organisations have adopted Inner Source, there is no clear insight into the key factors that must be considered in its adoption. Based on this observation, the researcher defined Research Question 6 as follows:

*RQ6: What are the key factors to consider in adopting Inner Source?*

Research Questions 5 and 6 together address the second research objective that led to the research presented in this thesis. Together they document challenges (RQ5) and provide insights into when and how Inner Source can thrive (RQ6).
Section 12.3 which follows summarises the contributions and implications of the results that this thesis makes.

12.3 CONTRIBUTIONS AND IMPLICATIONS FOR RESEARCH AND PRACTICE

This section presents the contributions and discusses the implications of the results for research and practice.

12.3.1 RQ1: Challenges in OSS

Chapter 6 presented the design and results of a systematic mapping study that has investigated the challenges related to the use of OSS components in product development. For this study, through a rigorous selection procedure to identify relevant studies, 17 papers were included in the review. A total of 21 different challenges were identified, which were classified into six categories. The most reported challenge was identifying quality products among the large supply of OSS products.

The results of RQ1 have a number of implications for researchers and practitioners. For researchers, the enumeration of challenges and classification may be a starting point for further research. It provides a “one-stop shop” offering a brief discussion of the various challenges as well as references to the original studies. Furthermore, researchers can use the findings for deliberating and debating the possible causes of the challenges, and suggest appropriate strategies and solutions for the identified challenges.

The findings may help practitioners to fully understand the potential OSS challenges and enable them to take appropriate measures to deal with them. A clear understanding of the potential risks may support organisations in the decision of whether or not to use OSS at all.

As mentioned, one of the key findings of the results presented in Chapter 6 is that selection and evaluation of suitable OSS products was found to be one of the most often reported challenges. The researcher had observed a number of OSS evaluation and selection methods proposed in
the literature. This led to RQ2, the contributions and implications of which are discussed next.

12.3.2 RQ2: OSS Evaluation methods

Chapter 7 investigated the state of the art of OSS evaluation methods. The chapter made three related sub-contributions:

- A literature review of OSS evaluation methods that have been proposed so far;
- A comparison framework to perform an analytical comparison of those methods;
- A comparison of six of the identified methods using the proposed framework.

The literature review identified 20 approaches to evaluate OSS products. These approaches were identified through a rigorous selection process, following the same procedure as for the systematic mapping study used in Chapter 6. In order to be able to systematically compare these approaches, a comparison framework was developed based on the literature. The framework, consisting of four components (context, user, process and evaluation) and a total of 13 comparison criteria, was then used to compare six methods.

The implications of the contributions for research and practice are as follows. Researchers may use this list to further investigate and analyse differences and commonalities for their particular research purpose. Chapter 7 also offered a framework that has been used to provide a systematic comparison of a number of these methods. The framework can be used and extended by other researchers, and the framework-based comparison may provide insights to researchers regarding the features and limitations of existing OSS evaluation solutions.

The overview of evaluation methods may inform practitioners of the proposed evaluation approaches, and the comparison framework may be used to help them to choose one that suits their needs.
12.3.3 RQ3: Architectural Knowledge needs of OSS integrators

Chapter 8 presented the design and results of an exploratory survey to address the question about the architectural knowledge (AK) needs of OSS components integrators. The study identified four types of AK that OSS integrators are interested in. The study also identified a number of reasons why they need such AK. Another finding of the study was that the availability of AK varies widely among OSS products. Furthermore, the study identified a few strategies to deal with a lack of AK.

The results of this study may provide input to other researchers to develop a research agenda with the aim to investigate this topic further; the researcher provides a few directions towards that goal in Section 12.4. In particular, since the study offers initial empirical evidence for the need of AK, the research community may be motivated to investigate how such AK can be identified and communicated from the OSS community to OSS integrators.

12.3.4 RQ4: Identifying Architectural Patterns in OSS

Chapter 9 presented the design and results of three studies that led to the definition and evaluation of a process for identifying architectural patterns (which is one type of AK) in OSS products. Following the Improvement Paradigm process, which is an iterative approach to improve processes, three empirical studies involving a total of 44 master’s students were conducted. The resulting process, named IDAPO, is a systematic, step-wise approach to gather information about an OSS product with the goal to identify architectural patterns. The process is aimed at guiding practitioners in this task. To evaluate the process, an experiment was conducted. The results of the evaluation indicated that, while there was no significant improvement in the number of identified patterns, the treatment group (that is, participants following the process) did identify more potentially present patterns (candidate patterns). This suggests that a step-wise approach for systematically gathering information of an OSS product can be helpful. However, the task of identifying architectural patterns remains to be extremely challenging.
Researchers may want to further investigate how the task of identifying patterns can benefit from both the heuristic approach that IDAPO proposes and the use of existing and new tools. Tools may be used as a means to extract information from the OSS product automatically, thereby supporting the manual activity of identifying and “matching” patterns. However, the study also revealed that existing tools are not always helpful; for instance, they cannot handle the source code structure (e.g., its organisation in a hierarchical folder structure).

12.3.5 RQ5: Challenges in Inner Source

Chapter 10 presented the design and results of an industrial case study at an organisation that had adopted a *project-based* Inner source model. Thirteen challenges were identified, ten of which could be mapped to an equivalent challenge in using OSS, as identified in Chapter 6. All 13 challenges could be classified into one of the six categories of OSS challenges presented in Chapter 6. Furthermore, a number of approaches were identified that the organisation had adopted to mitigate said challenges. For three challenges no mitigation approach was identified.

The study was conducted at an organisation that had adopted a project-based Inner Source model. It would be interesting which challenges occur in an infrastructure-based Inner Source model. Furthermore, the results came from a case study at one particular organisation. Since Inner Source implementations are always tailored to the organisation’s context, it would be interesting to conduct further studies at other Inner Source organisations.

For practitioners, these results may provide insights into the challenges that may occur when adopting Inner Source, as well as how to address some of these challenges.

12.3.6 RQ6: Key factors in adopting Inner Source

Chapter 11 presented a framework, grounded in the literature on Open Source and Inner Source, to assess organisations’ fitness to adopt Inner
Source. The assessment framework consists of 17 elements within four categories that were identified in the literature, and which may reveal potential tension points if an organisation decides to adopt Inner Source. In order to demonstrate the framework’s usefulness, an industrial case study was conducted at an organisation that had indicated an interest in adopting Inner Source.

The framework may be further validated and developed through empirical studies. As such, it may provide a starting point for further research. Organisations that are interested in adopting Inner Source may use it as a means to assess their fitness, or compatibility, with Inner Source.

12.4 FUTURE WORK

This thesis presents six contributions to the literature, which have been summarised in Section 12.3 as well as in Chapter 1 (see Figure 1-2). Based on these research results, the researcher suggests a number of directions for future work, which are discussed next. Since this thesis addresses two research objectives, resulting in two sets of research questions (RQ1-RQ4 and RQ5-RQ6), considerations for future work are presented along these two strands of research. Sub-section 12.4.1 presents future work regarding the use of OSS products, followed by Sub-section 12.4.2 which presents future work directions in research on Inner Source.

12.4.1 Future Work in Research on Using OSS products

Chapter 6 presented a set of challenges related to product development with OSS components. Chapters 7 to 9 subsequently focused on one challenge in particular, namely the evaluation and selection of suitable OSS products. Chapter 7 provided an overview of the various evaluation and selection methods that are available; however, research has also shown (as mentioned before) that practitioners do not typically use formal evaluation methods. Some open research questions that arise from this observation are:
• Why do practitioners not use such formal evaluation and selection methods?
• What are challenges in the evaluation and selection of OSS products?
• How do practitioners resolve such challenges encountered during the evaluation and selection of OSS products?
• Do existing evaluation and selection methods address these challenges?

As mentioned, Chapter 7 has provided an overview of existing evaluation methods to support the selection of appropriate OSS products. Each of these methods aims to gather a set of information about the OSS product, and the analysis suggested that there is little attention for a product’s architectural knowledge. Chapter 8 provided empirical evidence that practitioners would, in fact, appreciate architectural knowledge of an OSS product. Some open research questions that are worthwhile to investigate are the following:

• Do practitioners really need the information that is suggested by the various OSS evaluation methods?
• In which cases is architectural knowledge needed of OSS products (and when is it not important)?
• Besides architectural knowledge, what other types of information do practitioners need of an OSS product (that is not covered by the existing evaluation methods)?

Based on the findings presented in Chapter 8, the researcher developed a process (IDAPO), which is presented in Chapter 9. The goal of IDAPO is to guide practitioners in the task of identifying architectural patterns, which is one type of architectural knowledge. IDAPO was shown to be useful to a certain extent, but identifying architectural patterns remains to be a challenging task. It would be interesting to understand how experienced software developers and architects approach this task. Also, it seems it will be necessary to combine the use of tools to manage a large quantity of source code in a software product on the one hand, and manual approaches and heuristics on the other hand to understand in more detail which patterns are present in a product. Furthermore, while patterns may provide hints regarding the achievement (or non-achievement) of quality attributes, it would be interesting to understand to gain a deeper
understanding of the impact of patterns that support the selection decision. Some directions for future work are:

- How do experienced software developers and architects identify architectural patterns?
- How can tools effectively support the identification of architectural patterns?
- How can the impact of an identified architectural pattern be assessed in a way that is useful to make a selection decision?
- What are strategies to minimise architectural risks when integrating OSS components?

12.4.2 Future Work in Research on Inner Source

In contrast to Open Source, which has been studied for the last 15 years or so, Inner Source is very much a research area in its nascent phase, with only a small body of literature. Different Inner Source implementations seem to vary quite significantly; Alcatel-Lucent’s Inner Source initiative has resulted in quite an active internal “open source” project, whereas in Philips Healthcare it has proven quite difficult to build an active “community” or “bazaar” of contributors. In other words, some cases of Inner Source are more similar to “real” Open Source projects than others. It would be interesting to understand what are the factors that affect the vitality of an Inner Source community. Also, it may well be that many more organisations have adopted development practices that are similar to OSS development practices; however, so far these have not been reported as they have not been recognised as cases of “Inner Source”. As mentioned in Chapter 3, each organisation adopting Inner Source does so in its own way. Therefore, there is a certain continuum from “very open” to “somewhat open” Inner Source organisations. Identifying, studying and characterising more organisations that have adopted Inner Source would greatly help forward the Inner Source research field.

It is still an open question what factors influence the success of an Inner Source initiative; the study reported in Chapter 11 is a first step towards that goal. Furthermore, most of the published papers are general accounts of the Inner Source initiatives at various organisations (such as Hewlett-Packard, Alcatel-Lucent, and Philips). It would be extremely
interesting to conduct longitudinal studies at organisations that have adopted Inner Source to understand in more detail the motivations of developers to contribute. In particular, the researcher believes conducting longitudinal, ethnographic studies could provide much in-depth insight. Another interesting venue of research is to study in more detail the benefits of adopting Inner Source. Although Gaughan et al. (2009) have summarised a number of motivations and benefits that have been reported after adoption of Inner Source (see Chapter 3), these findings come from a relatively small number of interviews, and are not reported in much detail. Finally, the researcher believes that the Inner Source research field may draw lessons learnt by the Global Software Engineering (GSE) research area, and vice versa. In that context, it would be also quite interesting whether Inner Source initiatives can be effective in outsourcing and offshoring scenarios.

To summarise, the following are research questions that the researcher believes are worth investigating:

- Are there any other organisations that have adopted OSS development practices and how do they compare to the ones that have been reported so far?
- What are the key success factors for successful Inner Source adoption?
- What is the motivation of individual developers to contribute to a shared asset?
- Which OSS development practices can be effectively applied in commercial software development, and which are not suitable?
- What are the benefits of adopting Inner Source, and how do these manifest within organisations?
- Can Inner Source work in outsourcing/offshoring scenarios?

One of the key characteristics of OSS development is the non-monetary motivation of developers. As a result, there can be massively parallel efforts in, for instance, debugging and developing “prototypes”. Also, the constant reinvention to improve the design of the software is a key feature of OSS development. It seems unlikely that commercial software development organisations will ever support growing real OSS-style
bazaars in which different developers can implement the same software, or chase the same bug, in parallel. Perhaps the very nature of commercial software development, in which fixed release schedules and tight budgets are rule rather than exception, is the biggest obstacle to Inner Source.


Appendix A

Interview Guide used in Survey (Chapter 8)

This appendix contains the interview guide that was used for the survey reported in Chapter 8. The interview guide was loosely followed, as the interviews were semi-structured. The researcher depended heavily on follow-up questions on topics that emerged during the interviews.

• Your organization and position
  1. Size of organization
  2. Your position and how long
  3. Experience in software development?
  4. Involvement in how many projects that use OSS?

• Description of a recent/ongoing project (or projects) in which OSS was used
  1. Kind of product?
  2. Size of product?
  3. What were the main architectural issues/concerns in this project?
  4. How many OSS components were used?
  5. What kind of OSS components were used?

• Selection and Knowledge of OSS
  1. What architectural knowledge OSS products would you like to have, or do you need?
  2. Why do you need this knowledge?
  3. Was this knowledge available?
  4. If not, how do you deal with this?
  5. Compared to other factors, how important is architectural knowledge?
  6. What are the main evaluation criteria? How are OSS components selected? In the scientific literature there are a number of evaluation criteria/methods e.g. OSMM; do you use any formal selection process?
Concerns and challenges when using OSS components in products?
   1. What challenges and issues did you expect in using the OSS component?
   2. What challenges and issues did you experience?
   3. Main obstacle to use a certain OSS component?
Appendix B

Interview Guide used in Improvement Paradigm (Chapter 9)

This appendix contains the interview guide used in the first two iterations of the Improvement Paradigm method used in Chapter 9, which was followed in the design of the IDAPO process.

<table>
<thead>
<tr>
<th>Personal background</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. In what year did you enrol for your study Computing Science?</td>
</tr>
<tr>
<td>2. How much software design experience do you have?</td>
</tr>
<tr>
<td>3. How much programming experience do you have?</td>
</tr>
<tr>
<td>4. Do you do any programming in your spare time?</td>
</tr>
<tr>
<td>5. What programming languages do you use (not the languages you know, but actually use)?</td>
</tr>
<tr>
<td>6. Have you got any working experience as a software developer? If so, how long?</td>
</tr>
<tr>
<td>7. Have you got any experience as an Open Source developer?</td>
</tr>
<tr>
<td>8. If so, what project and how long have you been actively involved?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Knowledge</th>
</tr>
</thead>
<tbody>
<tr>
<td>9. Did you know the concept of design/architectural/software patterns before you started the course on Software Patterns?</td>
</tr>
<tr>
<td>10. If so, do you use software patterns when you develop software?</td>
</tr>
<tr>
<td>11. In your opinion, what effect does using software patterns have on the quality of software?</td>
</tr>
<tr>
<td>12. Do you have a particular interest in the topic Software Architecture? (in other words, do you find it an interesting topic?)</td>
</tr>
<tr>
<td>13. Do you have a particular interest in the topic of Software Patterns? (in other words, do you find it interesting?)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OSS Project</th>
</tr>
</thead>
<tbody>
<tr>
<td>14. What Open Source project did you choose to mine patterns?</td>
</tr>
<tr>
<td>15. What was the reason to choose that particular project?</td>
</tr>
<tr>
<td>16. Please estimate the size of this project.</td>
</tr>
<tr>
<td>17. Mining the architecture</td>
</tr>
<tr>
<td>18. Did you use any tools to analyse the architecture of the software?</td>
</tr>
</tbody>
</table>
19. If so, please specify them.
20. Did these tools help in analysing the architecture?
21. If so, please specify how.
22. What other means did you use to mine the architecture of the software?

**Mining patterns**

23. Please list the patterns that you have mined.
24. What variants of the patterns were used?
25. What quality attributes are affected by each pattern?
26. How are the patterns related?
27. Please estimate how many patterns the OSS project has.
28. Did you use any tools to mine *software patterns*?
29. If so, please specify them.
30. Did these tools help in mining patterns?
31. If so, please specify how.
32. What other means did you use to mine the patterns? Please indicate which ones, and how effective they were. (for instance, did you contact the OSS developers/community through mailing lists or IRC? If so, was that an effective means?)
33. Are you aware of any tools to help mine patterns?
34. What could be the requirements of such a tool?
35. What other kind of support tools or methods would be desirable for mining patterns?
36. In what ways, if any, did mining patterns help in recovering the architecture?
Appendix C

Interview Guide used in Case Study SoftCom
(Chapter 10)

This appendix contains the interview guide that was used in the case study at SoftCom, which was reported in Chapter 10. The guide also contains a number of questions that were used in the survey reported in Chapter 8.

• Your organisation and position
  1. Size of organisation
  2. Your position and how long
  3. Experience in software development?
  4. Involvement in how many projects that use OSS?

• Description of a recent/ongoing project (or projects) in which OSS was used
  1. Kind of product?
  2. Size of product?
  3. What were the main architectural issues/concerns in this project?
  4. How many OSS components were used?
  5. What kind of OSS components were used?

• Concerns and challenges when using OSS components in products?
  1. What challenges and issues did you expect in using the OSS component?
  2. What challenges and issues did you experience?
  3. Main obstacle to use a certain OSS component?

• Can you please tell something about Inner Source at SoftCom?
  1. What are OSS practices applied?
  2. What challenges have you encountered?
  3. What strategies or approaches have been put in place to overcome these challenges?

• Selection and Knowledge of OSS
  1. What architectural knowledge of OSS products do you need, or would you like to have?
2. Why do you need this architectural knowledge?
3. Is this knowledge typically available?
4. If not, how do you deal with this?
5. What are the main evaluation criteria? How are OSS components selected? In the scientific literature there are a number of evaluation criteria/methods e.g. OSMM; do you use any formal selection process?
Appendix D

Interview Guide used in Case Study NewCorp (Chapter 11)

This appendix contains the interview guide that was used for the interviews in the case study at NewCorp, presented in Chapter 11. The interviews were semi-structured, which means the interview guide was loosely followed. The researcher depended heavily on follow-up questions on topics that emerged during the various conversations.

<table>
<thead>
<tr>
<th>Introduction</th>
<th>1. Can you please describe your background, education, experience, position, and responsibilities?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2. Size of system currently working on; size of products in SLOC, number of hardware components.</td>
</tr>
<tr>
<td></td>
<td>3. What is the team size you’re working in?</td>
</tr>
<tr>
<td>Software design</td>
<td>1. Who is involved in architectural analysis and design?</td>
</tr>
<tr>
<td></td>
<td>2. What activities are performed to analyse the architecture?</td>
</tr>
<tr>
<td></td>
<td>3. What are the techniques and tools used during analysis?</td>
</tr>
<tr>
<td></td>
<td>4. What are the artefacts produced during architectural analysis, and where are they stored?</td>
</tr>
<tr>
<td></td>
<td>5. How much time does analysis/design take?</td>
</tr>
<tr>
<td></td>
<td>6. What are the main challenges during analysis?</td>
</tr>
<tr>
<td>Software development</td>
<td>1. What is the overall driver in the software development process?</td>
</tr>
<tr>
<td></td>
<td>2. What tools are used for software development? Does everybody have the same development environment/tools?</td>
</tr>
<tr>
<td>Open source software practices</td>
<td>1. What do you think is Inner Source? Can you give some example characteristics?</td>
</tr>
<tr>
<td></td>
<td>2. Which OSS Practices may be helpful in your daily software</td>
</tr>
</tbody>
</table>

335
development? For instance, giving commit access to many people in the company. How would this be helpful?

3. What current practices do you expect to be affected by these OSS practices?

4. Do you think using OSS practices will have an effect on the architecture, and would that be positive or negative?

5. How do you think Inner Source can address challenges in your current approach?