Identifying Architectural Patterns Used in Open Source Software: Approaches and Challenges

Klaas-Jan Stol[1], Paris Avgeriou[2] and Muhammad Ali Babar[3]
[1]Lero—The Irish Software Engineering Research Centre, University of Limerick, Ireland
[2]Department of Mathematics and Computing Science, University of Groningen, the Netherlands
[3]IT University of Copenhagen, Denmark
[1]klaas-jan.stol@lero.ie, [2]paris@cs.rug.nl, [3]malibaba@itu.dk

Background: Open Source Software (OSS) is increasingly used in product development. Besides some much-reported benefits of this approach, using OSS products also presents new challenges. One such challenge is identifying relevant, high-quality OSS products among the hundreds of thousands that are available. One approach for doing that is to identify architectural patterns, since these patterns have a direct effect on a product's quality attributes, such as performance and reliability. However, there are no well-defined methods or tools available to identify architectural patterns.

Research aim: Our goal is to identify approaches taken by novice software engineers that have no or little experience in identifying architectural patterns. We aim to get insight into how these novices tackle this problem, what challenges they encounter and what suggestions they have for improving this process.

Method: We collected data from seven M.Sc. student teams that performed a pattern identification assignment. We conducted semi-structured interviews with eight students from two teams. We studied reflection reports from four teams that reported their experiences as part of their final report. Furthermore, during his M.Sc. course, one of the authors performed the assignment as a member of a team. We also included his experiences.

Results and conclusions: We identified a number of approaches that students have taken in order to identify architectural patterns, as well as a number of challenges that they encountered in this task. Furthermore, based on suggestions from the students, we present a proposal to improve this process.

Keywords: Architectural patterns, pattern identification, approaches, challenges

1. INTRODUCTION

Software architectures of large scale systems influence the achievement of the desired quality attribute requirements (also called non-functional requirements) such as reliability, performance and maintenance. Hence, software architecture (SA) is considered an important artefact in the development and evolution of large scale software-intensive systems (Bass et al., 2003b). Software architectures of large systems are usually designed and implemented using suitable software patterns. One of the major purposes of using patterns is to develop software systems that are expected to provide the desired level of quality attributes (Bass et al., 2003a). In the context of our research, we use the word "pattern" to refer to architectural patterns or styles, as opposed to design patterns such as presented by the "Gang of Four" (Gamma et al., 1995). The software patterns community has discovered and documented dozens of patterns, for instance in (Avgeriou and Zdun, 2005; Buschmann et al., 1996). By identifying the patterns used in a system, it is possible to gain insight into the potential influence of those patterns on the system's quality attributes (Bass et al., 2003b). Another common observation is that large-scale software-intensive systems are not usually developed from scratch. Instead, component-based software development (CBSD) approaches have become a norm in developing large scale systems in the last decade or so (Szyperski, 1998).

Components may be built in-house or acquired from third parties. When acquiring third-party components, an organisation may purchase Commercial Off-The-Shelf (COTS) components, or decide to use Open Source Software (OSS) products. Indeed, OSS products are becoming more commercially viable (Fitzgerald, 2006). The use of OSS products in developing mission- and business-critical systems is becoming more and more an attractive area of research and practice with an ever-increasing number of available OSS products (e.g., since February 2009, Sourceforge (the largest repository of OSS projects, www.sourceforge.net) hosts more than 230,000 projects). Using OSS components instead of COTS components has several benefits, such as significantly lower costs (Fitzgerald, 2004), high product quality, availability of source code, and no vendor dependency.

However, many organisations may feel nervous about using OSS components which may not have been developed for achieving certain quality attributes (such as performance, reliability and security) or may have unpredictable consequences for the desired level of quality attributes of the overall system in which OSS components are integrated. Moreover, the integration efforts can face problems without access to the contextual information about the design decisions and assumptions about the kind of environment or architectures suitable for integrating those components (Bosch, 1999; Ali Babar and Gorton, 2007). Hence, it is quite challenging to evaluate an OSS product by accurately predicting the level of quality attributes to be achieved without access to the product's architectural documentation.

A promising approach to assessing the quality attributes supported or hindered by an OSS product is to identify its architectural patterns as they embody architectural design decisions that have a direct effect on the software's quality attributes (Harrison et al., 2007). However, it is quite difficult to identify and understand architectural patterns used in OSS for several reasons, such as lack of formalism for expressing patterns, difficulty in maintaining integrity of patterns' structure in large scale systems, and the coarse-grained nature of architectural patterns. Though several attempts have been made to manually and automatically identify design patterns, there has been no previous attempt to study the approaches and challenges involved in identifying architectural patterns used in OSS products.

We are investigating what approaches novice software engineers, that is, persons with little or no experience in industrial software development, apply to identify architectural patterns. Furthermore, we are interested in what challenges these novices encounter during the process. This will help us in formulating solutions to make the pattern identification process more straightforward. In this paper we report on an empirical study to gain insights into the pattern identification process followed by seven different teams of master's students. We identified approaches taken as well as challenges encountered. Furthermore, we report on the students' insights on how to improve the process of identifying architectural patterns. Based on these results, we propose a pattern identification process. This process may be suitable for assisting both students in the field of software engineering as well as practitioners that need to assess the quality of an OSS product. The remainder of this paper proceeds as follows. Section 2 presents background information. Section 3 presents our research design. Section 4 presents results, followed by a discussion in Section 5, while Section 6 concludes.

2. BACKGROUND

2.1. Open Source Software Evaluation

Several evaluation methods and frameworks have been proposed for assessment of the quality of OSS products. A few of these are: OpenBRR (Wasserman et al., 2006), OpenBQR (Taibi et al., 2007), QSOS (Atos Origin, 2006), Capgemini's Open Source Maturity Model (OSMM) (Duijnhouwer and Widdows, 2003) and Navica's OSMM (Golden, 2004). These evaluation methods typically assess the open source based on a number of criteria divided into a number of categories. Common categories are for instance "support", "product quality", "documentation", "community", and "maturity". In most methods each criterion is assigned a certain weight, and the final score is a weighted average of the assessment scores. While such factors as the maturity and level of support certainly help in assessing the quality of an OSS product, it does not provide insight into the architectural quality of a product nor does it provide insight into the consequences for the quality attributes of the overall system in which the OSS products are integrated.

2.2. Pattern Identification

Various tools have been proposed for design pattern identification; an overview is presented in (Dong et al., 2007). Design patterns are software patterns, but without architectural significance. That is, they do not affect the quality attributes of the system as a whole, although Avgeriou and Zdun note that design patterns can potentially be used as architectural patterns (Avgeriou and Zdun, 2005). Furthermore, design patterns, such as presented in (Gamma et al., 1995) are object-oriented, which assumes that the software is written in an object-oriented language. This is fundamentally different from architectural patterns in an SA. An SA consists of a number of components, which are typically more coarse-grained than classes and objects. A component can be as small as a single class or object (such as a language parser) or as big as a complete subsystem, such as the Apache web server. There are no commonly accepted formalisms for describing components and connectors between them. Proposed formalisms, such as various Architecture Description Languages (ADLs) and the UML have issues (Medvidovic and Taylor, 2000). Zhu et al. have demonstrated that software pattern identification can be used to extract architecturally important information (Zhu et al., 2004). This information can then be used to identify the quality attributes, which will help the developer to assess the quality of a product, or at least highlight some potential points of concern. Paakki et al. presented a tool called "Maisa" to do architectural and design pattern mining from UML diagrams (Paakki et al., 2000). Of course, this approach is based on the assumption that UML diagrams are available. Maisa assumes that an object-oriented implementation language is used for the construction of the software under investigation. Although the paper claims that Maisa is suitable for architectural pattern identification as well, the paper does not demonstrate that.

3. RESEARCH DESIGN

3.1. Research questions

We are interested in identifying approaches that novice software engineers would take in the task of identifying architectural patterns. This will give us insight into how this task is typically done by practitioners that are not specially trained in this task. Hence our first research question, which we will answer in Section 4.1:

RQ1: What are the approaches that participants take to identify architectural patterns?

Architectural pattern identification is a complex task, since these patterns are more difficult to identify in the source code than design patterns (in particular, object-oriented design patterns). However, we lack insight in what challenges may arise in the pattern identification task. Understanding these challenges may help us in formulating solutions to make the task more straightforward. Our second research question is thus:

RQ2: What challenges did participants encounter during architectural pattern identification?

We will answer RQ2 in Section 4.2. The aim of this research is to increase our understanding of how to do pattern identification in an effective way. We are therefore very much interested in how this process can be improved, hence our third research question that we will address in Section 5 is:

RQ3: How can the process of identifying architectural patterns be improved?

3.2. Data collection methods

We have gathered experiences from M.Sc. students who have identified architectural patterns in OSS products that are under active development and of considerable size. In total seven different teams performed a pattern identification assignment, varying in size from two to six students. All teams were free to select a non-trivial OSS product themselves, which had to be approved by the instructors before starting on the assignment. No instructions were given on how to perform the assignment. All students were doing a course in computer science, and all but one were specialising in software engineering. Most students had some practical experience in industry, such as a part-time software development job. The students had various nationalities including China, Germany, India, Ireland, Italy, Luxembourg, the Netherlands and Tanzania.

For this research we gathered data from three different sources. An overview of our data sources is presented in Table 1. The first column indicates the data source as well as when the data was acquired, whereas the column “Date studied” indicates when the assignments were performed. The column “Team size” indicates the number of students in the team; if specified, the number between brackets indicates the number of students participating in our research (i.e. from whom we received feedback). If no bracketed number is specified, input was received from all team members.

Table 1: *Data sources for this research.*

Data source	Date studied	OSS Project	Location	Team size
semi-structured interviews (Feb. ’09)	*January 2009*	*JBoss*	*Univ. of Groningen*	*6 (5)*
		Eclipse	*Univ. of Groningen*	*6 (3)*
Reflection reports (May ’09)	*April 2009*	*MeDiCi*	*University of Limerick*	*3*
		Filezilla	*University of Limerick*	*3*
		HackyStat	*University of Limerick*	*3*
		ServiceMix	*University of Limerick*	*3*
Author’s Experience (Sep. ’09)	*January 2007*	*Parrot*	*Univ. of Groningen*	*2 (1)*

3.2.1. Interviews

We invited 12 students at the University of Groningen in the Netherlands to participate in semi-structured interviews. Although conducting interviews is resource demanding, interviews are highly interactive, which allows the researcher to ask for clarification to gain a deep understanding of the topic (Lethbridge et al., 2005). The students had taken the course “software patterns” in 2009, given by one of the authors. Most of them have no or limited work experience. Students were asked to identify at least five architectural patterns. The students were given four weeks for the assignment. We asked 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. All interviews were digitally recorded with the participants’ consent. Five of these interviews were with Dutch students, and were conducted in their mother tongue since the interviewer is a native speaker. The other three interviews (with students from Germany, India and Tanzania) were done in English. The recorded interviews were transcribed, and the interviews conducted in Dutch were translated into English by the interviewer. In order to prevent that anything was “lost in translation” the translations were checked against the original recordings.

3.2.2. Reflection reports
Our second source of data was the experiences reported by another group of master's students that had done architectural pattern identification. The assignment was one of three assignments in a course "software architecture", given in 2009 by one of the authors. Students were asked to identify at least five architectural patterns. Students were 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 in Ireland. Five teams of three students performed the pattern identification assignment. They were asked to explicitly report their experiences and reflections on the pattern identification process. One team did not provide any reflection on their experiences in their report, which is why we did not include that report for our data collection.

3.2.3. Author's experience
The third source of data was the first author's experiences of doing architectural pattern identification. This was done as an assignment of a master's course in "software patterns" at the University of Groningen, given in 2007, given by one of the authors of this paper. The assignment 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. Four weeks were given to finish this assignment. We note that this author is a contributor to the OSS project that was studied ("Parrot"), and has therefore a thorough understanding of that project.

3.3. Data analysis

We analysed the data as follows. The transcripts of the interviews were read, and phrases that were considered relevant to our research questions were highlighted using different colours for different topics ("approach", "challenge", etc.), so as to be able to quickly identify the topic of the phrase. The reflection reports were analysed in a similar fashion. The first author of this paper noted down a number of approaches and challenges that he encountered in his assignment. All data were entered into a database together with the source (filename and page number or timestamp for interview transcripts). This allowed for easy cross-checking and tracing back of the data to the source by all researchers.

The first author classified approaches by labelling each one of them. For example, a certain approach may be to "read the documentation"; we labelled this approach as "documentation". Through this process, a set of categories emerged. For each label a justification was provided to support the choice of the label. The second and third researchers then checked the results of the initial classification. Any disagreements were resolved through discussion. After the labelling step, we grouped approaches based on their label to identify equivalent approaches and challenges, which could then be unified into a single approach or challenge. Challenges were related to approaches during which they were encountered.

4. RESULTS

4.1. Approaches for Identifying Patterns

Our analysis resulted in a list of 20 approaches that the students have taken. We listed related approaches into a number of categories that emerged during data analysis. Table 2 presents the approaches that we have identified. The remainder of Section 4.1 discusses these approaches in more detail.

Table 2: *Approaches taken by students to identify patterns.*

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

4.1.1. Software under investigation
We 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).

4.1.2. Documentation

We 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 cross-check diagrams (A7). We 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). We labelled this as "documentation" (as opposed to "source code") since students would first study the documentation and then cross-check that with the source code.

4.1.3. Source code and tools

We 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). We 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.

4.1.4. Components and connectors

We 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).

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).

4.1.6. Miscellaneous

We identified two approaches that did not seem to fit in any other category, which is why we classified them 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).

4.2. Challenges in identifying architectural patterns

We identified 29 different challenges that students encountered during the assignment, which are listed in Table 3. We group the identified challenges according to the respective approaches (as identified in Section 4.1). We note that not all approaches have associated challenges. This grouping will help us to gain insight in what problems may arise as a result of taking a certain approach. Such understanding can assist us in defining a set of guidelines to make pattern identification a more straightforward process. The remainder of Section 4.2 discusses these challenges in more detail.

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.

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.

4.2.3. Approach A3: challenges

Three 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).

4.2.4. Approach A6: challenges

We 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).

Table 3: *Challenges encountered during pattern identification.*

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

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.

4.2.6. Approach A9: challenges

We 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.

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).

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).

4.3. Improving the pattern identification process

We asked the eight interviewees to give us suggestions about improving the pattern identification process. In our formulation of the questions we asked specifically for ideas that could help us in designing a tool or a pattern identification method. We present the results from the analysis of the responses to this question in Table 4. We categorised the suggestions in a similar way as the approaches in Section 4.1.

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, 2004). 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).

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.

Table 4: *Suggestions for improving the pattern identification process.*

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

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).

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).

5. DISCUSSION AND LIMITATIONS

5.1. A Pattern Identification Process

Since architectural pattern identification cannot be fully automated, it is important to provide a procedural set of guidelines that may help practitioners to identify patterns. In this section, we propose such a process that can assist both students and practitioners to identify patterns. The process is based on our empirical findings; the steps that we defined are based on the participants' approaches and enriched with the participants' suggestions. As we defined the various steps, we have considered the various challenges in order to provide guidance to the process' users. We refer to approaches and suggestions from Section 4 in the descriptions of the various steps of the process so as to indicate the relation to those empirically identified approaches and suggestions. We note that we do not include any use of tools in our process, which is why we do not refer to approach A13 and suggestions S10-S12. We emphasise 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. Our proposed process is shown in Figure 1, using UML activity diagram notation.

We assert 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, 2004), the Open Pattern Repository (Manteuffel and Verspai, 2009) and PAKME (Ali Babar and Gorton, 2007). The process is discussed in detail in the following subsections.

5.1.1. 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).

5.1.2. 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.

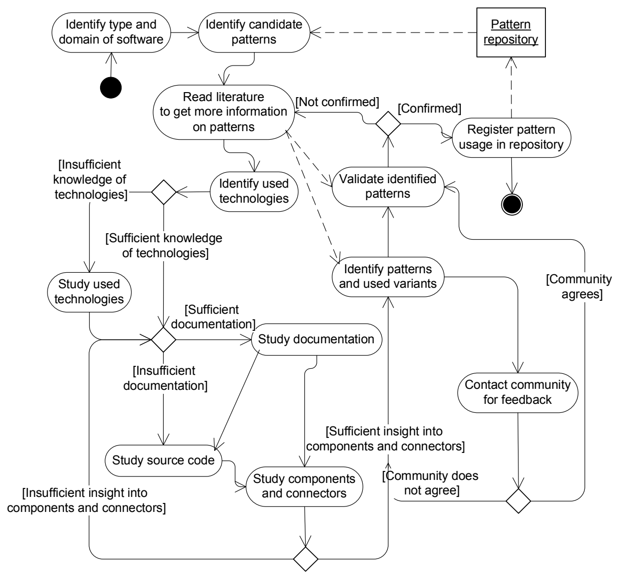
Identify type and domain of software
Identify candidate patterns
Pattern repository
Read literature to get more information on patterns
[Not confirmed]
[Confirmed]
Register pattern usage in repository
[Insufficient knowledge of technologies]
Identify used technologies
Validate identified patterns
[Sufficient knowledge of technologies]
Study used technologies
Identify patterns and used variants
[Community agrees]
[Sufficient documentation]
Study documentation
[Insufficient documentation]
Contact community for feedback
[Sufficient insight into components and connectors]
Study source code
[Community does not agree]
[Insufficient insight into components and connectors]
Study components and connectors

Figure 1: *Our proposed pattern identification process.*

5.1.3. Read literature to learn about patterns
We do not expect 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.

5.1.4. 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).

5.1.5. 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.

5.1.6. Study documentation
If there is sufficient documentation available, we propose 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.

5.1.7. 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.

5.1.8. 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, we argue to have a number of iterations to go back to the documentation and source code, to verify or adjust these insights.

5.1.9. 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).

5.1.10. Contact community for feedback

After finding the patterns, we propose 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), we argue 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.

5.1.11. 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.

5.1.12. 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, we suggest to go back to study the literature on patterns in order to gain a better understanding of patterns.

5.2. Limitations of this study

We are aware of a few of limitations of our study, which we discuss next. Firstly, the interviews that we have conducted were transcribed and translated by a single researcher. 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. Also, since the translation was not checked by the other researchers, this may have introduced errors in the data. However, we think that this risk is minimal for two reasons. Firstly, we think the researcher who did the translations has sufficient knowledge of the English language to do a correct translation. Secondly, after translation, the translated transcripts were checked by listening once more to the digital recordings of the interviews to make sure that no information was lost. The second limitation of this research is that it is unlikely that we have captured all approaches and challenges for identifying architectural patterns. However, this field of study is still very young, and we therefore feel that the exploratory nature of this research is a useful contribution. A third limitation is the potential bias of the third data source (author's experiences) as his results were noted down after collecting data from the other participants. However, the goal of our research is to gain insight into approaches and challenges of pattern identification, and we feel this does not influence the validity of our findings in this exploratory research. Furthermore, 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 (A8). On the one hand this can be classified as "documentation"; on the other hand this may be classified as "source code". We resolved any of these disagreements through discussion.

6. CONCLUSIONS AND FUTURE WORK

Using Open Source Software (OSS) components in product development is recognised to be a viable option for software developing organisations. While there are many high-quality OSS products available, identifying high-quality products is a challenging task. We believe that identifying architectural patterns can help in evaluating the quality of OSS products. However, identifying architectural patterns is a challenging task by itself and has received little research attention. This situation motivated us to investigate this process in closer detail. We gathered data from M.Sc. students, who can be considered novice software engineers, i.e. practitioners with limited experience of software engineering in general and architectural patterns in particular. We asked them about their experiences of identifying architectural patterns in OSS products. We collected data from eight semi-structured interviews and four reflection reports. Furthermore, we included experiences of the first author of this paper, who has performed a pattern identification assignment as part of his M.Sc. course. This paper has two main contributions. Firstly, we have empirically identified approaches taken, and challenges experienced by students through different data sources. Furthermore, we collected ideas for improving the pattern identification process. Secondly, based on these empirical findings, we have proposed a process that practitioners may follow in order to perform pattern identification in a more systematic way. We believe this process can be of help particularly for novice software engineers with limited experience in software design and architecture. We plan to continue our research in the following directions. The second and third authors of this paper will continue teaching software architecture and patterns classes. We are designing empirical studies that will be embedded in these modules based on the results we presented in this paper. Specifically, we will present our proposed process for identification of patterns in these classes, and empirically assess the value of the process. Based on this evaluation we plan to further refine and improve this process and develop appropriate tooling support for automating as many steps of the proposed process as possible.

ACKNOWLEDGEMENTS

This work is partially funded by IRCSET under grant no. RS/2008/134 and by Science Foundation Ireland grant 03/CE2/I303_1 to Lero (www.lero.ie). We are grateful to the students who participated in this research. Thanks go out to Ewoud Werkman for his contribution to the pattern identification assignment performed together with the first author of this paper.

REFERENCES

Ali Babar, M. & Gorton, I. (2007) A Tool for Managing Software Architecture Knowledge. *Proceedings of the Second Workshop on SHAring and Reusing architectural Knowledge Architecture, Rationale, and Design Intent.* IEEE Computer Society.

Atos Origin (2006) Method for Qualification and Selection of Open Source software (QSOS) version 1.6.

Avgeriou, P. & Zdun, U. (2005) Architectural Patterns Revisited - a Pattern Language. *Proceedings of the 10th European Conference on Pattern Languages of Programs (EuroPLoP 2005).*

Bass, L., Bachmann, F. & Klein, M. (2003a) Deriving Architectural Tactics-A Step toward Methodical Architectural Design. Software Engineering Institute, Carnegie Mellon University.

Bass, L., Clements, P. & Kazman, R. (2003b) *Software Architecture in Practice,* Boston, MA, USA, Addison-Wesley.

Booch, G. (2004) Handbook of software architecture. http://www.handbookofsoftwarearchitecture.com accessed on: December 15, 2009.

Bosch, J. (1999) Product-line architectures in industry: a case study. *Proceedings of the 21st international conference on Software engineering.* Los Angeles, California, United States, ACM.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. & Stal, M. (1996) *Pattern-oriented Software Architecture - A System of Patterns*, J. Wiley and Sons Ltd.

Dong, J., Zhao, Y. & Peng, T. (2007) Architecture and Design Pattern Discovery Techniques - A Review. *International Conference on Software Engineering Research and Practice.*

Duijnhouwer, F. & Widdows, C. (2003) Open Source Maturity Model. *Capgemini Expert Letter.*

Fitzgerald, B. (2004) A Critical Look at Open Source. *Computer,* 37**,** 92-94.

Fitzgerald, B. (2006) The transformation of open source software. *Management Information Systems Quarterly,* 30**,** 587-598.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995) *Design patterns: Elements of Reusable Object-Oriented Software,* Reading, Massachusetts, Addison-Wesley.

Golden, B. (2004) *Succeeding with Open Source*, Addison-Wesley.

Harrison, N. B. & Avgeriou, P. (2008) Analysis of Architecture Pattern Usage in Legacy System Architecture Documentation. *Seventh Working IEEE/IFIP Conference onSoftware Architecture (WICSA 2008)*

Harrison, N. B., Avgeriou, P. & Zdun, U. (2007) Using patterns to capture architectural decisions. *IEEE software,* 24**,** 38-45.

Lethbridge, T. C., Sim, S. E. & Singer, J. (2005) Studying Software Engineers: Data Collection Techniques for Software Field Studies. *Empirical Software Engineering,* 10**,** 311-341.

Manteuffel, C. & Verspai, M. (2009) Open Pattern Repository. http://code.google.com/p/openpatternrepository accessed on: December 7, 2009.

Medvidovic, N. & Taylor, R. N. (2000) A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering,* 26**,** 70-93.

Paakki, J., Karhinen, A., Gustafsson, J. & Nenonen, L. (2000) Software metrics by architectural pattern mining. *Proceedings of the International Conference on Software: Theory and Practice.*

Szyperski, C. (1998) *Component software: beyond object-oriented programming*, Addison-Wesley.

Taibi, D., Lavazza, L. & Morasca, S. (2007) OpenBQR: a framework for the assessment of OSS. IN Feller, J., Fitzgerald, B., Scacchi, W. & Sillitti, A. (Eds.) *Open Source Development, Adoption and Innovation.* Springer.

Wasserman, A. I., Pal, M. & Chan, C. (2006) The Business Readiness Rating: a Framework for Evaluating Open Source.

Zhu, L., Babar, M. A. & Jeffery, R. (2004) Mining patterns to support software architecture evaluation. *Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA'04).*