IDAPO: A Process for Identifying Architectural Patterns in Open Source Software

Klaas-Jan Stol and Paris Avgeriou

November 15, 2011

1 Introduction

Software systems are usually designed and implemented using certain architectural patterns, or styles. Even if a software designer or developer is not aware of certain patterns, the patterns may still be present! Architectural patterns, such as layers and model-view-controller (MVC) have a certain effect on the system's quality attributes (QAs), such as performance and reliability. A large number of architectural patterns have been documented, in for instance the Pattern-Oriented Software Architecture (POSA) series of books (e.g., (Buschmann *et al.*, 1996)). Well-known is also the book by the "Gang of Four" (GoF) (Gamma *et al.*, 1995), which documents a number of object-oriented design patterns. Although design patterns are implemented at a finer level of granularity, design patterns may also be implemented at an architectural level. In general, there is no hard line between the two types of patterns (Avgeriou and Zdun, 2005). For each pattern it is documented what *forces* it resolves, and what are the *consequences* of using the pattern. Thus, knowledge of which architectural patterns are used within a system provides some insight into the system's quality attributes.

However, information about which architectural patterns (or just "patterns" from hereon) is often not available, in particular of Open Source Software (OSS), since documentation about such architectural design decisions (ADDs) (i.e., the use of certain patterns) is often lacking in OSS projects.

For software developers, knowledge of which architectural patterns can be quite valuable, since it may hint at the potential (in)compatibility of an OSS product with the system in which it will be integrated.

This document is structured as follows: Section 2 provides a brief overview, followed by Section 3, which presents each step of IDAPO in detail.

2 Overview of IDAPO

This section provides an overview of IDAPO. Previously, we reported on the development, design and evaluation of IDAPO in (Stol *et al.*, 2010) and (Stol *et al.*, 2011). This document aims to provide a more in-depth and detailed description of IDAPO.

The purpose of IDAPO is to provide a systematic approach to gather information about an OSS product that may be relevant to the task of identifying architectural patterns.

Table 1 presents a brief description of each step, as well as input and output expected for each step.

Table 1. Overview of steps, inputs and outputs.

Step	Description	Input	Output
1	Identify the type and domain of the product	N/A	Type and domain information
2	Identify used technologies	N/A	Used technologies
3	Study used technologies	Used technologies	
4	Identify candidate patterns	Used technologies, type and domain information	Candidate patterns
5	Read patterns literature	Candidate patterns	List of patterns and participants
6	Study documentation	List of patterns and participants	List of main components
7	Study source code	Documentation, source code, list of main components	N/A
8	Study components & connectors	List of main components, documentation, source code	Components & connectors
9	Identify patterns and variants	Source code, documentation, patterns literature, components & connectors	Identified patterns
10	Validate identified patterns	Identified patterns, Patterns literature	Validated patterns
11	Get feedback from community	Identified patterns	N/A
12	Register pattern usage	Validated patterns	N/A

Figure 1 shows the IDAPO process using Business Process Modelling Notation (BPMN). BPMN is a standard notation with well-defined semantics. For an introduction to the BPMN language, please refer to (White, 2004).

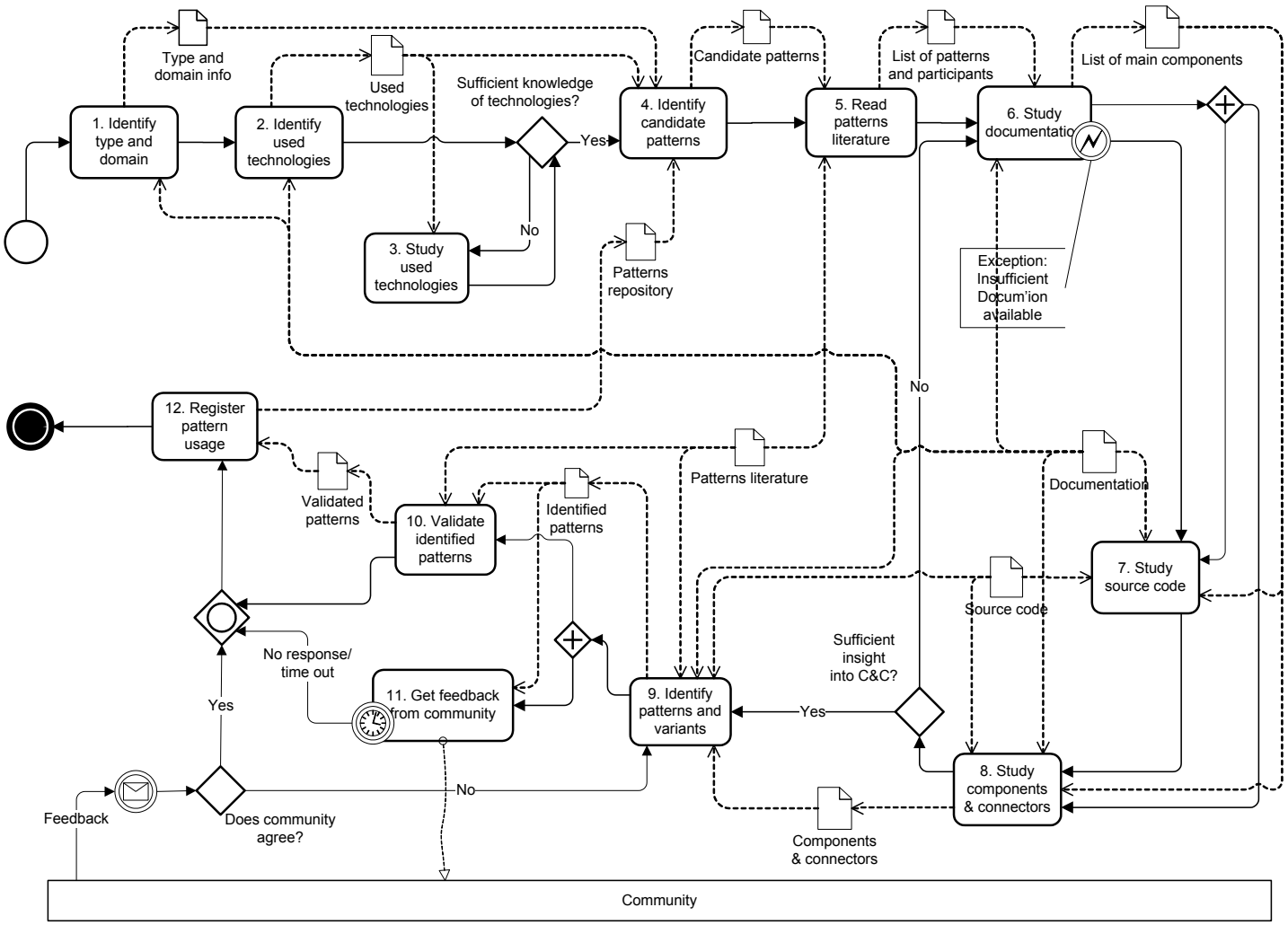

Figure 1. IDAPO: A Process for Identifying Architectural Patterns in Open Source Software (in BPMN notation).

Section 3 which follows next provides a more detailed description of each step, inputs and outputs.

3 Steps of IDAPO

IDAPO consists of 12 steps. This section presents each step in detail.

3.1 Identify type and domain of software

The first step is to identify the type and domain of the software under investigation. The "type" of software refers to a category of software. For instance, type could be "instant messaging", "web server", or "content management system" (CMS). The "domain" refers to the field of application in which the software would be used, and what its main functionality is; for instance, a software product could be in the "file management" domain, in which case its functionality is for instance to manage files (think of DropBox). There is no fixed set of types and domains to choose from, though the classification used on www.sourceforge.net may be of help.

After performing this step, you should have insight into the type of system and its domain. In other words, what type of system is it, what does it do, and how would you use it?

3.2 Identify used technologies

The second step is to identify used technologies of the software. A technology could be for instance a certain framework, such as Struts (for web-based Java applications). This may hint at the presence of certain patterns (such as MVC). Another example of a technology is CORBA, which is a standard for Object Request Broker (ORB) software. More information on technologies and patterns can be found in (Mahdavi Hezavehi *et al.*, 2011).

After performing this step, you should have insight into the technologies used in the software.

3.3 Study used technologies

In case you don't have sufficient knowledge of one of the identified technologies, step 3 is to find more information about this. For instance, if your knowledge of CORBA is rusty (and you identified it as a used technology), you could read up on this technology. If you know enough about the used technologies, you may skip this step.

After performing this step, you should have sufficient knowledge of the used technologies that are used in order to be able to understand what role they play in the system, and how they were used.

3.4 Identify candidate patterns

Based on insights gathered from the first three steps, the fourth step is to identify *candidate* patterns. A candidate pattern is a potentially used or present pattern. For instance, if you identified that the Struts[1] framework was

[1] http://struts.apache.org/

used, you may suspect that the MVC pattern was used. This is a candidate pattern, because at this stage you may not be 100% sure that it is really there. Nevertheless, the pattern should be listed as potentially present.

After performing this step, you should have a list of potentially present patterns, and also an idea where in the system it is present. This need not be very precise; a rough estimate is sufficient.

3.5 Read literature to learn about patterns

From the previous step you have a list of candidate patterns. In order to be able to confirm that these patterns are really there, you may need to consult the patterns literature. For instance, the book by the "Gang of Four" (GoF) (Gamma *et al.*, 1995), and the POSA book series (e.g., (Buschmann *et al.*, 1996)). The purpose of this step is to gain better insight into the details of the patterns, which will help in turn to assert whether the pattern is really present.

After performing this step, you should have a good understanding of each pattern that you identified as a candidate pattern.

3.6 Study documentation

The next step is to study the project documentation. In particular, look for information that may provide hints at patterns. In some cases, patterns are already explicitly mentioned and explained, depending on the project. Otherwise, try to identify the main components that are present in the product, and the connectors between the components (in other words, how do the components communicate?). This step may also contribute to the list of candidate patterns; that is, during this step, you may identify other candidate patterns that you had not listed before.

After you performed this step, you should have a clear insight into what components the product is composed of, and how the various components communicate (the connectors).

In exceptional cases, there is no documentation available at all. In most cases there will be some documentation, but it may happen that none is available. In that case you can skip this step.

After this step, proceed to both step 7 and 8; you can perform these steps in an alternating fashion; that is, study both the source code, compare findings to your findings from step 8 (study C&C), and so on.

3.7 Study source code

The purpose of Step 7 is to study the source code. In particular, the goal is to try to confirm the information you have gathered so far in the source code. That means that you would try, for instance, to identify the components identified in the previous step (see Section 3.6) in the source code. Also, the source code may contain hints about the use of certain patterns, for instance through naming conventions, or in documentation (comments) embedded in the source code.

This step should be performed in parallel with step 8 (see Section 3.8).

After performing this step, you should be able to identify which components are located where in the source code. What source code elements (source code files, packages (in case of Java source code), and so on) map to which components (and connectors)? Also, this step may contribute to the list of candidate patterns.

3.8 Study components and connectors

Step 8 is to study components and connectors. In particular, based on the "list of components" resulting from step 6 (see Section 3.6), the goal would be to identify these components in the source code (see Section 3.7).

The purpose of this step is to identify components and connectors. You may have to go back to Step 6 (study documentation), and also to step 7 (study source code). This is an iterative process.

After this step you should have a clear understanding of the components and connectors in the product. What components are there, and how do they communicate?

3.9 Identify ("match") patterns

Once you have identified the components and connectors (see Steps 7 and 8), it is time to try to confirm which patterns of the "candidate patterns list" are present. During this step, you can consult the patterns literature (e.g., GoF, POSA) to make sure that you have really identified a candidate pattern.

After you performed this step, you should have progressed from a "candidate" patterns list to a "identified patterns list". Also, which variant of the pattern did you identify? (for instance, in case of Pipes-Filter, is it really a pure Pipes-Filter, or is it perhaps a Batch-Sequential pattern?)

After this step, you should proceed with steps 10 and 11 in parallel. Step 10 is to validate the identified patterns, for instance by letting a group member double check the presence of a pattern.

3.10 Validate identified patterns

Step 10 is to validate the identified patterns. Preferably, you would let another group member identify the same patterns, or let him/her confirm the presence of the pattern. By doing so, you have a "second opinion" of whether the patterns are really there. In case you have disagreements, you should resolve these through group discussions.

3.11 Contact community for feedback

Step 11 is to contact the OSS community. While you are of course free to contact the community at any time, OSS developers have limited time, and are more likely to respond when presented with something, rather than respond to a "cold call" in which you ask them for information. If you have shown to make an effort, community members are more likely to provide feedback. You can contact the community through its mailing list.

If the community agrees with your findings, you can proceed with Step 12. If the community does not agree, you should go back to Step 9. It may happen that you don't get a response from the community. You could try to ask the same question again, but if you don't get a response at all, just proceed with the next step (Step 12).

3.12 Register usage of patterns

The last step of the process is to register the patterns you identified (and validated) in a repository. This way, other people can consult the information that you gathered. One such repository is the Open Pattern Repository (van Heesch, 2010). Alternatively, you may report your findings in a technical report. Please consult with your course instructor how you should report your findings.

4 References

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

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

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

Mahdavi Hezavehi, S., van Heesch, U. & Avgeriou, P. (2011) A Pattern Language for Architecture Patterns and Software Technologies: Introducing Technology Pattern Languages, Proceedings of the 16th European Conference on Pattern Languages of Programs (EuroPLoP). http://www.hillside.net/europlop/europlop2011/submission/shepherd.cgi?token=7974c292b70044d1ee00995a6c033093f33734db&action=download&label=1309356706_25

Stol, K., Avgeriou, P. & Ali Babar, M. (2010) Identifying Architectural Patterns Used in Open Source Software: Approaches and Challenges, *Proceedings of the 14th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, Keele, UK.

Stol, K., Avgeriou, P. & Ali Babar, M. (2011) Design and Evaluation of a Process for Identifying Architecture Patterns in Open Source Software, in: Crnkovic, I., Gruhn, V. & Book, M. (Eds.) *Proceedings of the 5th European Conference on Software Architecture, Essen, Germany*. Springer-Verlag Berlin Heidelberg, LNCS vol. 6903, doi:10.1007/978-3-642-23798-0_15, pp. 147-163.

van Heesch, U. (2010) Open Pattern Repository, http://www.cs.rug.nl/search/ArchPatn/OpenPatternRepository

White, S.A. (2004) Introduction to BPMN, BPTrends, http://www.zurich.ibm.com/~olz/teaching/ETH2011/White-BPMN-Intro.pdf