Towards an Automated Pattern Selection Procedure in Software Models

Alexander van den Berghe, Jan Van Haaren,
Stefan Van Baelen, Yolande Berbers, and Wouter Joosen

{firstname.lastname}@cs.kuleuven.be
IBBT-DistriNet, Department of Computer Science, KU Leuven
Celestijnenlaan 200A, 3001 Leuven, Belgium

Abstract. Software patterns are a widely adopted technique to manage the rapidly increasing complexity of software. Despite their popularity, applying software patterns in a software model remains a time-consuming and error-prone manual task. This paper proposes a novel approach to the automated selection of applicable patterns. We argue that software models and patterns are relational, and propose using relational learning to assign general roles to software model elements, which are being used to select the most appropriate patterns. Furthermore, our approach provides hints on how to instantiate these patterns in the software model.

Keywords: Relational Learning, Software Pattern Selection, Logic Programming, Application

1 Introduction

The complexity of both software and the software development process has increased rapidly over the past decades because of three major reasons. The first reason is the steadily increasing complexity of the problems to be solved by software. The second reason is the shift towards distributed software, which entails a number of additional issues to account for. The third reason is the relatively long lifetime of software, which is often much longer than that of the hardware it was originally developed for and requires it to adapt to an ever changing environment.

A number of techniques have been proposed to manage the complexity of software. Software patterns provide an established solution to recurring issues in software development [3, 1] and thus improve the overall quality, portability and readability of a software design. Although software patterns are widely adopted by software developers, applying patterns remains a mostly manual two-step task. First, a developer selects the most appropriate pattern based on his or her

This research is partially funded by the Flemish government institution IWT (Institute for the Promotion of Innovation by Science and Technology in Flanders), by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, and by the Research Fund KU Leuven.

previous experiences, which is an increasingly difficult and time consuming task due to the steadily growing number of patterns. Second, a developer instantiates this pattern in the software design, which is a repetitive and error prone task.

In this paper, we propose a novel approach that automates the selection of applicable patterns and supports the instantiation of patterns in a software design. We argue that both software designs and software patterns are highly relational and that logic programming is well-suited to reason about them. Our approach relies on a concise relational representation of both software models and software patterns whereas current pattern selection approaches require an extensive description of the design problem.

2 Background on Software Engineering

This section provides the essential background on software development and software patterns for this paper.

2.1 Software Development

On a very high level, designing software boils down to gathering requirements from the end users and ensuring these requirements are satisfied in the final software system. Typically, software developers define a number of components, which each satisfy a subset of the requirements, in order to manage the complexity of software. Each component offers its functionality through one or more interfaces, which other components can make use of.

Software developers capture design decisions more formally in graphical models, showing the different components and how their interfaces interact with each other. Furthermore, they annotate each component with additional information (e.g., implementation related details) that is required during software development.

2.2 Software Patterns

Software patterns offer established solutions to recurring design issues and comprise four elements. The first element is a unique name. The second element generically describes the design issues that the pattern resolves. The third element generically describes the solution that the pattern proposes (i.e., independent of any concrete design or implementation decision). The fourth element provides the consequences and trade-offs of the pattern, which are helpful when deciding on the most appropriate pattern.

Automatically selecting applicable patterns, requires a formal representation of either the problem or solution description. Our approach relies on a formalization of the solution description, which is achieved with Zdun and Avgeriou's concept of primitives [9]. These primitives, which have precisely defined semantics, are building blocks for software patterns. Each pattern combines several primitives to offer a solution to a specific design issue.

Figure 1 shows the Client-Dispatcher-Server pattern comprising the Client, Dispatcher and Server primitives [1].

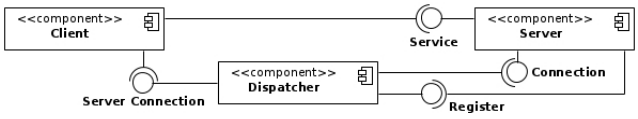

Fig. 1. Model for the Client-Dispatcher-Server pattern. Rectangles represent components and circles represent interfaces. A line between them denotes which component offers an interface and a line ending in half a circle denotes which component uses it.

3 Related Work

Pattern selection has only been given little attention in academic literature. Kampffmeyer and Zschaler [6] use a pattern intent ontology that can be queried with design issues and return all applicable patterns. Kim and El Khawand [7] first formally model each pattern as a set of roles and then determine which patterns are applicable by verifying which roles the software model fulfills. Hsueh et al. [5] introduce a goal-driven approach that proposes applicable patterns by asking relevant questions to the developer. Hasheminejad and Jalili [4] first classify the patterns and the design issue using text classification techniques and then propose the best matching patterns from the design issue's class.

Current pattern selection techniques require either an extensive specification of the design problem (e.g., [6, 5, 4]) or the patterns' problem descriptions (e.g., [7]), which is often cumbersome and time-consuming, and considerably limits their usability and reliability. Besides, it is not obvious how to extend these techniques to new (types of) patterns since either an extensive analysis is required or the terminology in their description should be picked carefully. Furthermore, most techniques provide no hints on how to instantiate the applicable patterns in the software model such that instantiating patterns remains a manual task.

4 Approach

We propose Automated Role Based Pattern Selection (ARBPS), a novel approach to automated pattern selection in software models. Software models define a number of components and show how these components collaborate whereas software patterns show how certain components should collaborate to solve a specific design issue. Therefore, both software models and software patterns are naturally represented in a relational way as a set of objects (i.e., components) and relations among these objects (i.e., collaborations), which ARBPS achieves using the logic programming language Prolog.

ARBPS has three main steps. The first step concerns formally representing the available software patterns in a way that allows reasoning about them. The second step encompasses assigning one or more roles to each component in the

software model. The third step involves selecting applicable patterns, given the formal representation of the available patterns and the roles in the software model. A pattern is applicable if it fulfills all roles of the considered components.

In the following, we will discuss each of these three steps in further detail. We will use the software model in Figure 2 as a running example throughout this discussion. The software model represents a small digital newspaper system, which lets clients browse and read through news articles, and subscribe themselves to specific categories of news and is based on [8].

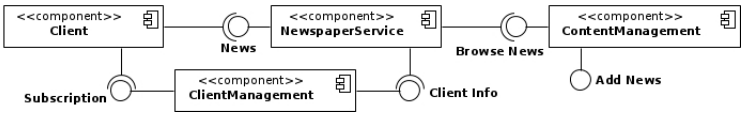

Fig. 2. Running example of a digital newspaper system.

4.1 Representing Software Patterns

The first step concerns formally representing the available software patterns as a set of primitives, where each primitive fulfills one or more roles. ARBPS represents the relevant information as a Prolog knowledge base that consists of two classes of facts. The first class of entity facts enumerates the available patterns, primitives and roles. The second class of relation facts defines relations between patterns, primitives and roles. These relations express which primitives each pattern comprises, and which role(s) each primitive fulfills, where each role must be fulfilled by at least one primitive. Knowing which element in a primitive fulfills which role(s), allows ARBPS to provide hints on how to instantiate an applicable pattern in the software model.

For the running example, we add entity facts for the Client-Server-Dispatcher pattern, the primitives Client, Dispatcher and Server, and the roles Service Provider, Service Interface and Service Requester. Furthermore, we add, amongst others, relation facts expressing that the Client-Server-Dispatcher pattern consists of the primitives Client, Dispatcher and Server, and that the Server component of the Server primitive fulfills the role of the Service Provider.

4.2 Assigning Roles in the Software Model

The second step encompasses assigning one or more roles to each component in the software model, which is a time-consuming and error-prone endeavor. Therefore, we propose using machine learning to (partially) automate this procedure and thus reduce the workload of software developers significantly. Software models and software patterns are relational such that we can naturally represent them in the statistical relational learning framework kLog [2]. kLog is a language for kernel-based relational learning that builds upon several simple but powerful concepts such as learning from interpretations, data modeling through entities and relationships, deductive databases and graph kernels.

Representing a software model in kLog boils down to defining an entity for each component and a relationship for each collaboration between two components (i.e., one component using the interface of another component). Each entity has a number of properties, which represent the requirement(s) that the corresponding component accomplishes in the software model. Furthermore, each entity has a class label that denotes the role(s) that the corresponding component fulfills. A class label refers to either a single role or a set of related roles.

The key idea is to leverage the expert knowledge and previous efforts of software developers to assign roles to components in a software model. Each software model corresponds to a single interpretation (i.e., tuples that are true in an example), where previously annotated software models are training interpretations. The design problem at hand is the test interpretation, which possibly already contains class labels for a number of entities (e.g., assigned by a software developer). The learning task is predicting the class labels of the remaining entities.

The main challenge is modeling the requirements that a component accomplishes in a software model. Software developers employ the responsibilities of a component and the components it collaborates with to determine its role(s) in a software model. In recent years, a number of dedicated languages have been proposed to formally represent requirements. Currently, we are investigating which of these languages is best suited for our setting, and how we can intuitively model these formal representations in kLog.

Therefore, we assume in the remainder of this paper that a software developer has already assigned roles to the components in the software model. For the running example the Service Provider role is assigned to the NewspaperService and ClientManagement components. The Service Requester role is assigned to the Client component.

4.3 Selecting Applicable Software Patterns

The third step involves selecting applicable patterns by mapping the roles of the components in the software model to the roles of the available patterns. ARBPS verifies for each available pattern whether it fulfills all of the roles annotated to the software model. A pattern is applicable if and only if each of these roles is fulfilled by at least one of its primitives. The task in this step thus boils down to querying the knowledge base we defined in the first step with the roles we learned in the second step.

ARBPS relies on two main queries to retrieve all applicable patterns from the knowledge base. The first query returns for a given pattern and a given role which of the pattern's primitives fulfills this role or fails when no primitive is found. For the running example, this query returns the Server primitive when called with the Client-Dispatcher-Server pattern and the Service Provider role. The second query returns a list of applicable patterns for a given set of roles by iteratively calling the first query for any possible combination of an available pattern and a given role. For the running example, this query returns the Client-Dispatcher-Server pattern when called with the roles from the second step.

Besides offering a list of applicable patterns, ARBPS also provides hints on how to instantiate these patterns in a software model. Calling the first query for each applicable pattern and each of the roles in the software model, enables to establish a precise mapping between the software model elements and the pattern elements.

5 Conclusion and Future Work

We have introduced a novel approach to the automated selection of applicable software patterns in software models. Our approach relies on a concise relational representation of both software patterns and software models, which enables reasoning about software patterns and automatically assigning roles to software model elements. Furthermore, our approach easily allows for the inclusion of additional patterns and provides hints on how to instantiate the selected patterns in a software model.

The automated assignment of roles to software model elements is our main research direction. The key challenge is finding an expressive as well as intuitive way of representing formal requirements in the relational learning framework. Furthermore, we are investigating how to score applicable patterns such that we can order them and thus limit the number of proposals. Otherwise, software developers might still spend a lot of time selecting the most appropriate pattern from an extensive list of proposed patterns.

References

1. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture: A System of Patterns. Wiley (1996)
2. Frasconi, P., Costa, F., De Raedt, L., De Grave, K.: kLog: A Language for Logical and Relational Learning with Kernels (2012), *arXiv:1205.3981v3*
3. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional (1994)
4. Hasheminejad, S.M.H., Jalili, S.: Design Patterns Selection: An Automatic Two-Phase Method. The Journal of Systems and Software 85(2), 408–424 (Feb 2012)
5. Hsueh, N.L., Kuo, J.Y., Lin, C.C.: Object-Oriented Design: A Goal-driven and Pattern-based Approach. Software and Systems Modeling 8(1), 67–84 (2009)
6. Kampffmeyer, H., Zschaler, S.: Finding the Pattern You Need: The Design Pattern Intent Ontology. In: Model Driven Engineering Languages and Systems. pp. 211–225. No. 4735 in Lecture Notes in Computer Science (2007)
7. Kim, D.K., El Khawand, C.: An Approach to Precisely Specifying the Problem Domain of Design Patterns. Journal of Visual Languages & Computing 18(6), 560 – 591 (2007)
8. Van Landuyt, D., Op de beeck, S., Truyen, E., Verbaeten, P.: Building a Digital Publishing Platform Using AOSD. In: LNCS Transactions on Aspect-Oriented Software Development. vol. 9, pp. 1–34 (December 2010)
9. Zdun, U., Avgeriou, P.: Modeling Architectural Patterns Using Architectural Primitives. ACM SIGPLAN Notices 40(10), 133–146 (Oct 2005)