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
iMinds-DistriNet, Department of Computer Science, KU Leuven
Celestijnenlaan 200A, 3001 Leuven, Belgium

Abstract. Software patterns are widely adopted 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. In this paper, we argue that the relational nature of both software models and software patterns can be exploited to automate this cumbersome procedure. First, we propose a novel approach to selecting applicable software patterns, which requires only little interaction with a software developer. Second, we discuss how relational learning can be used to further automate this semi-automated approach.

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 reasons. The first reason is the steadily increasing complexity of the problems that software tackles. 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 which requires it to adapt to an ever changing environment.

Software patterns provide established solutions to recurring issues in software development [5, 1] and hence 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 patterns based on his or her previous experiences. This is an increasingly difficult and time-consuming task

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.

due to the steadily growing number of patterns. Second, a developer instantiates these patterns in the software design, which is a repetitive and error-prone task.

In this paper, we argue that the relational nature of both software models and software patterns can be exploited to automate this cumbersome procedure. We apply inductive logic programming techniques for representing these models and patterns as well as reasoning about them. First, we propose a semi-automated approach to selecting applicable patterns in a graphical software model. Our approach relies on a concise relational representation of both the available software patterns and a software model. The approach requires only little interaction with a software developer whereas current pattern selection procedures rely on an extensive specification of the design problem. Second, we briefly share ideas on how relational learning techniques can further automate our approach and hence reduce the amount of user interaction.

2 Background on Software Engineering

On a very high level, designing software is gathering requirements and ensuring these requirements are met in the final software system. Software developers typically identify a number of components, which each satisfy a subset of the requirements, in order to manage the complexity of software. Components offer their functionality through one or more interfaces via which they collaborate.

Software developers formally capture their design decisions in graphical models. Each model element can be annotated with additional information (e.g., implementation details) that is required during the development process. Although our approach is applicable to multiple types of software models, we restrict ourselves to the Component-and-Connector Model, which is one of the most important types of models during software development. Figure 1 shows an excerpt of a digital newspaper system, inspired by [10], in which clients can browse through news articles, read news articles and subscribe to specific categories of news.

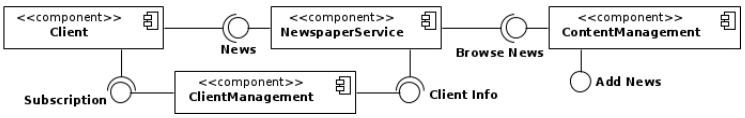

Fig. 1. Software model of a digital newspaper system where rectangles represent components and circles represent interfaces. A line denotes a component offering an interface whereas a line ending in half a circle denotes a component using an interface.

Software patterns offer established solutions to recurring design issues. Alongside a unique name, a software pattern comprises a generic description of both the design issue it addresses and the solution it proposes (i.e., independent of any concrete design and/or implementation decision). Furthermore, a software

pattern contains an overview of its consequences and trade-offs, which is helpful to decide on the most appropriate software pattern.

Automatically selecting applicable software patterns requires a formal representation of either the design issue a software pattern addresses or the solution it proposes. Our approach leverages Zdun and Avgeriou's concept of primitives [12] to formally represent a pattern's proposed solution. Primitives have precisely defined semantics and can be seen as building blocks for patterns. A software pattern combines several primitives to offer a solution to a specific design issue.

Figure 2 shows the `Client-Dispatcher-Server` pattern, which comprises the `Client`, `Dispatcher` and `Server` primitives [1].

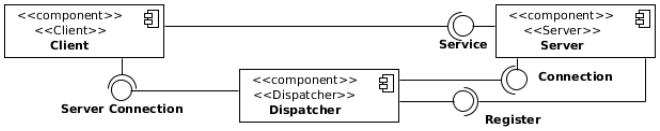

Fig. 2. Software model of the Client-Dispatcher-Server pattern.

3 Related Work

Pattern selection has only been given little attention in academic literature. Kampffmeyer and Zschaler [8] use a pattern intent ontology that can be queried with design issues and return all applicable patterns. Kim and El Khawand [9] first formally model each pattern as a set of pattern-specific roles and then determine which patterns are applicable by verifying which roles the software model fulfills. Hsueh et al. [7] introduce a goal-driven approach that proposes applicable patterns by asking relevant questions to the developer. Hasheminejad and Jalili [6] first classify the patterns and the design issue using text classification 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., [8, 7, 6]) or the patterns' problem descriptions (e.g., [9]), 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 chosen carefully.

Although automated pattern selection has only been given little attention to date, the idea of tackling software engineering issues using machine learning is not new. One area of interest is that of software pattern detection, where the task is to identify applied software patterns when re-engineering legacy software. Correa et al. [2] perform this task using Prolog as a representation language. Another area of interest is that of software testing, where the task is to evaluate how well software systems meet their specification [3, 11].

4 Contributions

This section discusses the two main contributions of this paper. First, we propose Semi-Automated Role-Based Pattern Selection (SARBPS), a novel semi-automated approach for selecting applicable software patterns in software models. The approach exploits the relational nature of both software patterns and software models to significantly reduce the time required to decide which software patterns are applicable in a given software model. Second, we discuss how relational learning can be used to further automate our approach and hence reduce the amount of user interaction.

4.1 Contribution 1: Semi-Automated Role-Based Pattern Selection

We now discuss the three steps of the SARBPS approach in turn. The **first step** concerns representing the available software patterns as sets of primitives, where each primitive fulfills one or more roles. SARBPS represents this information as a knowledge base consisting of two classes of facts. The first class is the class of *entity facts* that enumerate the available patterns, primitives and roles. The second class is the class of *relation facts* that define relations between patterns and primitives on one hand and primitives and roles on the other hand. These relations express which primitives each pattern comprises and which role(s) each primitive fulfills. Roles are only meaningful if fulfilled by at least one primitive.

The **second step** involves annotating the components and interfaces of the software model with additional information. A software developer employs the requirements that the components and interfaces satisfy to assign one or multiple *roles*, which describe their properties and characteristics. Roles indicate which application-independent function the annotated element fulfills in a software system. For example, a component that provides services to other components can be assigned the `ServiceProvider` role. Only roles defined in the knowledge base constructed in the previous step can be assigned.

The **third step** involves selecting applicable patterns by establishing a mapping between the roles of the elements in the software model on one hand and the roles of the available patterns on the other hand. For each of the available patterns, SARBPS verifies whether it is applicable. A pattern is applicable if and only if each of the software model's roles is fulfilled by at least one of the pattern's primitives. SARBPS achieves this by querying the knowledge base it constructed in the first step using the roles assigned during the second step.

SARBPS uses two 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. 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.

Example 1. In order to illustrate our approach, we use the software model and software pattern shown in Figures 1 and 2 respectively. Due to space constraints, we only provide a high-level discussion in this paper.[2] In the **first step**, we add the `Client-Dispatcher-Server` pattern and the three primitives this pattern comprises, `Client`, `Dispatcher` and `Server`, to the knowledge base. We also add the roles `ServiceProvider` and `ServiceRequester`, which the `Server` and `Client` primitives fulfill respectively. In the **second step**, we assign the `ServiceProvider` role to both the `NewspaperService` and `ClientManagement` components, and the `ServiceRequester` role to the `Client` component. In the **third step**, querying the knowledge base for applicable software patterns returns the `Client-Dispatcher-Server` pattern.

Although our semi-automated pattern selection approach saves significant amounts of precious development time, manually assigning roles to components and interfaces is still an error-prone and time-consuming task, especially for large, real-world software systems. Therefore, we propose using relational learning to automate this task and share some ideas on how to do this in what follows.

4.2 Contribution 2: Towards Automated Role Annotations

When annotating a component or an interface with one or multiple roles, software developers leverage the descriptions of the requirements each component or interface fulfills. These descriptions are mostly free text but they tend to have a simple structure. For example, one of the requirements in our running example states that a journalist must be able to add an article to the system. In a medical system, a requirement could be that the medical staff must be able to add a new patient file to the system. Although the descriptions of these requirements are completely different, it is clear that they can be written as instances of the same abstract pattern since they share a common structure.

The missing building block is a convenient formal language for representing software requirements. Once we have such a language, we can naturally model requirements in a relational learning system and pose the assignment of roles to components and interfaces as a collective classification task. We can then represent components and interfaces as *entities*, and collaborations amongst these components and interfaces as *relations*. The classification task boils down to learning one or multiple roles for each component. The key idea is to leverage the expert knowledge and previous role labeling efforts of software engineers, who can possibly already assign roles to some of the components and interfaces.

The kernel-based relational learning framework kLog [4] is a suitable learning system to perform this task. An important advantage of kLog is that the outcome of its *graphicalization phase*, a ground entity-relationship model, is conceptually very similar to a software model and hence easily interpretable by a software engineer. Preliminary results on a toy example yield encouraging results.

[2] A thorough discussion of the running example and a Prolog implementation are available at `https://people.cs.kuleuven.be/alexander.vandenberghe/sarbps.html`.

5 Conclusion and Future Work

We have introduced a novel semi-automated approach for selecting applicable software patterns in software models, which makes developers to save significant amounts of development time. Our approach relies on a concise relational representation of both software patterns and software models, which allows reasoning about them and including additional software patterns in a straightforward way.

Our main research direction is further automating the approach using relational learning techniques. The key challenge is designing a convenient formal language for representing software requirements in a relational learning framework. Furthermore, we aim to limit the number of proposed patterns by incorporating each pattern's trade-offs and consequences.

Acknowledgments. We thank Paolo Frasconi and Luc De Raedt for providing access to the kLog implementation.

References

[1] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture: A System of Patterns. Wiley (1996)
[2] Correa, A., Werner, C., Zaverucha, G.: Object Oriented Design Expertise Reuse: an Approach Based on Heuristics, Design Patterns and Anti-Patterns. In: on Heuristics, Design Patterns and Anti-patterns, in Proceedings of the 6th International Conference on Software Reuse. pp. 336–352 (2000)
[3] DeMillo, R.A., Offutt, A.J.: Constraint-Based Automatic Test Data Generation. IEEE Trans. Softw. Eng. 17(9), 900–910 (Sep 1991), http://dx.doi.org/10.1109/32.92910
[4] Frasconi, P., Costa, F., De Raedt, L., De Grave, K.: kLog: A Language for Logical and Relational Learning with Kernels (2012), *arXiv:1205.3981v3*
[5] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional (1994)
[6] 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)
[7] 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)
[8] 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)
[9] 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)
[10] 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)
[11] Vanmali, M., Last, M., Kandel, A.: Using a neural network in the software testing process. International Journal of Intelligent Systems 17(1), 45–62 (2002), http://dx.doi.org/10.1002/int.1002
[12] Zdun, U., Avgeriou, P.: Modeling Architectural Patterns Using Architectural Primitives. ACM SIGPLAN Notices 40(10), 133–146 (Oct 2005)