Tool Support for Refactoring to Design Patterns

Carmen Zannier
University of Calgary, Computer Science
Calgary, Alberta, Canada, T2N 1N4
(00) (1) (403) 220 7140
zannierc @cpsc.ucalgary.ca

ABSTRACT
Using design patterns improves the maintainability of software systems. Applying patterns often implies upfront design while Agile Methods rely on software architecture to emerge. We bridge this gap by applying refactoring towards patterns to improve software design. We propose complex refactoring to J2EE design patterns and describe requirements of complex refactorings and accompanying tool support.

Keywords
Refactoring to Design Patterns; J2EE Design Patterns; Atomic Refactorings; Sequential Refactorings; Complex Refactorings;

1. INTRODUCTION
Design patterns enhance the readability, maintainability and understand-ability of a software system [5]. They usually require the use of software development methodologies that implement thorough upfront design. Agile methodologies emphasize an initial, but emerging software design and architecture [2]. We join these two approaches by proposing complex refactorings to design patterns and accompanying tool support that allow design to emerge after a software system has been coded, but still conform to a given design pattern. Complex refactorings are composed of atomic and sequential refactorings and maintain design knowledge of an existing system. We focus on refactoring to Java 2 Enterprise Edition design patterns, a popular application area. The benefits of complex refactorings and tool support therein are those typical of design pattern implementations as well as improved run-time performance. Section $2^{@}$ examines refactoring to design patterns, Section 3 discusses the required design pattern knowledge, Section 4 explains the benefits and desired goals and Section 5 concludes what we have discussed.

2. REFACTORING TO PATTERNS
The increased popularity of Agile methods such as Extreme Programming, Scrum, Crystal, etcetera has helped advertise a design and code improvement practice: refactoring. At the very basic level, refactoring is cleaning up code. Small changes to code such as renaming a variable preserve the behavior of a software system while restructuring the code to improve readability and understand-ability [4]. Such atomic refactorings are as basic as a refactoring can be and tool support for such operations is available [3][7]. At a similar level to these atomic refactorings are refactorings such as **ExtractMethod**, which moves a section of (possibly repeated) code into its own method [4]. These refactorings are only slightly more complex than the atomic refactorings previously mentioned and thus are considered atomic. Tool support for these refactorings is also easily found. The complexity of refactorings quickly increases with the combination and repetition of atomic refactorings. An example of this is **ExtractClass**, which encompasses **MoveField** and **MoveMethod** [4]. We term such refactorings sequential as they are sequences of atomic refactorings. Tool support for sequential refactorings is not as easily found. Finally, an even higher complexity level is determined. Made up of atomic and sequential, complex refactorings lean towards introducing patterns into a system. An example is **Replace Multiple Constructors with Creation Methods**, a work in progress by Kerievsky [8] which involves a combination of **ExtractClass**, and a repetition of **RenameMethod** and **MoveMethod** atomic and sequential refactorings. These complex refactorings are necessary to handle the complexity of changing the entire structure of a software system and are the focus of this research.

The complex refactorings we propose use J2EE design patterns as targets. We address four issues. Firstly, what knowledge do refactorings require? Atomic and sequential refactorings maintain little to no knowledge of an application's design. Renaming or moving a method requires little information (names of methods in the class, the name of another class) about the actual structure of an entire application. Complex refactorings produce changes in an application at a higher abstraction level and require more knowledge of an application's design. Fortunately, our complex refactorings will be used to refactor to design patterns, each of which has a determinable structure that provides information for the complex refactorings. Similarly, the motivation behind applying a pattern is somewhat determinable. The complex refactorings represent knowledge of the motivation and the structure of the solution. We use Session Façade as an example. The motivation behind the use of the Session Façade pattern is typically tight coupling between the client and the business tier. For example, a Java Server Page (.jsp) directly accesses an entity bean. This creates much dependence between the two layers so that a change in one layer affects the other layer a great deal. In order to enhance the application, complex refactorings must know where the client pages (e.g. a .jsp) reside and where the business pages (e.g. entity beans) reside. The complex refactorings must also know that any references to entity beans within the .jsp pages need to be changed to reference the session façade class. Complex refactorings require much more knowledge than atomic or sequential refactorings and our tool support will reflect this.

The second issue is what atomic or sequential refactorings, or combinations of such refactorings should be used within each complex refactoring? For example, in order to refactor to the Session Façade design pattern there are a few tasks that must be performed, assuming the existence of a weaker design such as the

OOPSLA '02, November 4-8, 2002, Seattle, Washington, USA.
ACM 1-58113-626-9/02/0011.

motivation given above. 1. Create a new session bean façade class to communicate between the client and business objects; 2. Instantiate the business objects inside the façade; 3. Instantiate the Session Façade inside the client; 4. Move and rename much of the functionality out of and within (respectively) the client. We establish the complex refactoring **Create Conversation Class** which encompasses creating the actual Session Façade class, instantiating it in the client and instantiating business objects in the Session Façade class. The atomic and sequential refactorings involved are Fowler's **ExtractClass**, which encompasses **MoveField** and **MoveMethod** and we add the atomic command to instantiate the facade class (the business objects can be instantiated with **MoveField**). Each complex refactoring will therefore be made up of its requisite knowledge of the system design and a number of atomic and sequential refactorings.

The third issue is what existing tool support can we utilize to assist in development of the tool? The refactoring support provided by IDEs such as [3][7], is support for atomic and sequential refactorings only. These refactorings are behavior preserving, undoable and extremely useful for simple tasks, but they require much user involvement and do not change the structure of the entire system to a large extent. To order and combine just these atomic and sequential refactorings to refactor to a design pattern would require much organization and foresight by the user, and would risk the stability of the program. Eclipse is an open source IDE developed by IBM and contains constantly growing atomic and sequential re-factoring support [3]. Like similar tools, it is currently insufficient to refactor to a design pattern, but was chosen for its availability of code and potential for refactoring expansion. Eclipse contains refactoring wizards, refactoring classes and change classes to support atomic and sequential refactorings, all of which can be used to manipulate low-level refactorings within the complex refactorings. Any necessary but yet unimplemented atomic and sequential refactorings will be implemented according to the existing refactoring design, based on the design of [10]. The final issue is what complex refactorings can be used or established to enhance runtime performance? We focus on design patterns at the J2EE Business Tier, which address many network call issues, and focus on complex refactorings to these design patterns.

3. PATTERN CREATOR KNOWLEDGE

In our tool, each Design Pattern Creator Wizard maintains a Knowledge Store specific to its design pattern. The Knowledge Store contains Motivation Knowledge and Solution Knowledge. The Motivation Knowledge represents the typical errors ideally solved by the given design pattern. The Solution Knowledge represents the desired structure of the application. For example, in the Session Façade Design Pattern Creator Wizard, the Motivation Knowledge contains information about a series of client pages and a series of business objects. The Solution Knowledge contains information about the Session Façade, the business objects and the client. By including this knowledge in the tool we reduce the number of necessary requests to the user.

4. BENEFITS AND DESIRED GOALS

The benefits of tool support for refactoring to J2EE Design Patterns are many. By refactoring the code we allow for improved naming of variables and methods, enhancing code readability and code understand-ability. In conforming to a given design pattern we improve understand-ability of an application as a whole and we improve the flexibility of an application [5][6]. Equally important though is our desire to improve the development process and run-time performance. Trying to manually refactor a reasonable sized application to the extent that a design pattern is adopted is tedious and difficult. With tool support however, we simplify and speed up the process by hiding some of the necessary but time-consuming atomic refactorings. Lastly, we improve run-time of the applications. We focus specifically on problems in calls across the network and try to perform these calls in the most optimal manner possible, as per the given design pattern.

5. CONCLUDING REMARKS

Traditional software development favors sound up-front design. Agile software development favors emerging design. We propose complex refactorings to join these conflicting approaches by allowing design to change even after an application is implemented, regardless of whether the design was established up-front or emerged throughout the development process. The accompanying tool combines atomic and sequential refactorings to create complex refactorings with design pattern motivation and solution knowledge that restructure the design of an entire application. The domain is J2EE applications and the desired goal is improvement in the following areas: readability, flexibility, understand-ability, development process and run-time.

6. REFERENCES

[1] Alur D, Crup J, Malks D; Core J2EE Patterns Best Practices and Design Strategies; Sun Microsystems Inc. Upper Saddle River, NJ; 2001; p.54-71, 104-112, 246-420.

[2] Beck, K.; Extreme Programming Explained: Embrace Change; Addison Wesley Upper Saddle River NJ, 2000; p.103-115.

[3] Eclipse www.eclipse.org (Last Visited: July 11, 2002).

[4] Fowler, M.; Refactoring: Improving the Design of Existing Code; Addison-Wesley ; Upper Saddle River, NJ 2000; p. xv-xxi, 110-116, 142-153, 227-231, 273-274.

[5] Gamma E. Helm R. Johnson R. Vlissides J; Design Patterns – Elements of Reusable Object Oriented Software; Addison Wesley; Reading MA, 1995; p.1-3

[6] Grand, Mark; Java Enterprise Design Patterns, Patterns in Java 3; John Wiley & Sons Inc. New York 2002; p1-6.

[7] IntelliJ IDEA www.intellij.com/idea (Last Visited: July 11, 2002).

[8] Kerievsky, Joshua; Refactoring to Patterns; Industrial Logic www.industriallogic.com/papers/rtp015.pdf (Last Visited: July 11, 2002).

[9] Opdyke W.; Refactoring Object-Oriented Frameworks; PHD Dissertation, University of Illinois at Urbana-Champaign 1992.

[10] Roberts, D, Brant J, Johnson R; A Refactoring Tool for SmallTalk; Journal of Theory and Practice of Object Systems (TAPOS) V.3 No.4 1997 p.253-263.