Tool Support for Complex Refactoring to Design Patterns

Carmen Zannier[1], Frank Maurer[1]

[1] University of Calgary, Department of Computer Science,
Calgary, Alberta, Canada T2N 1N4
{zannierc, maurer}@cpsc.ucalgary.ca

Abstract. The abstract should summarize the contents of the paper and should Using design patterns is seen to improve 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 complex refactoring towards patterns to improve software design. Complex refactorings are based on existing tool-supported refactorings, knowledge of the application to be changed, knowledge of design patterns, and the capability to generate necessary code for a given design pattern. We present complex refactorings to J2EE design patterns and describe requirements of complex refactoring and accompanying tool support.

1 Introduction

Design patterns enhance the readability, maintainability and flexibility of a software system [6]. They usually require the use of software development methodologies that implement thorough upfront design. A conflict exists when examining agile methodologies which emphasize an initial but emerging software design and architecture, and rely on tacit knowledge of said design and the YAGNI (You Ain't Gonna Need It) principle [3]. Agile methods assume that creating more flexibility than is currently needed is wasted effort while design patterns are used to increase the flexibility of software for anticipated changes.

While agile teams generate immediate feedback in the form of programmed functionality, [3], the question as to the level of comprehension each developer has of the system remains unanswered. Contrastingly, while traditional methodologies provide a basis for a developer's design knowledge in the form of design documentation, the question of the team's ability to satisfy customer requirements is prolonged until late in the life of the project. In an attempt to profit on the strengths of these conflicting approaches we ask: can we utilize agile methodologies and still create "good" design? That is, can we have sound emerging design? We propose complex refactoring to design patterns and accompanying tool support that helps developers change the design of existing software to conform to a design pattern more typically found in top-down development methodologies. We present complex refactorings,

comprised of atomic and sequential refactorings, and which maintain design pattern knowledge, initial application design knowledge and the capability to generate code. As examples, we focus on complex 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 implementation as well as improved runtime performance. The goal of this research is to specify requirements for a tool that helps developers refactor to a given design pattern and to provide a proof of this concept. Section 2 gives some background information, Section 3 examines the definition of Refactoring, Section 4 discusses the tool's knowledge, Section 5 looks at an example, Section 6 specifies our contributions and we conclude with a look at the current state, future work and a final summary.The preparation of manuscripts which are to be reproduced by photo-offset requires special care. Papers submitted in a technically unsuitable form will be returned for retyping, or canceled if the volume cannot otherwise be finished on time.

2 Background

The Gang of Four, [6], initially defined design patterns as "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context". We focus on design patterns found in the business tier of Java2 Enterprise Edition applications. Some concerns that are addressed with these patterns in [2] are: tight coupling between the client and business tier, resulting in a proliferation of network calls; mapping the object model directly to the entity bean model; mapping each use case to a session bean and embedding service lookups in clients. The example found in Section 5 deals with the Value Object pattern and the Session Façade pattern.

3 Refactoring

The increased popularity of agile methods such as Extreme Programming, Scrum, Crystal, [1], has helped advertise a design and code improvement practice: refactoring. At the very basic level, refactoring is cleaning up code while preserving the behaviour of an application [5]. We group existing refactorings into two categories and present a third, more complex category.

3.1 Atomic Refactoring

Small changes to code such as renaming a variable, improves the readability and of software [5]. We term these 'atomic refactorings' as they are primitive refactorings comprised of only one or two operations (e.g. change name of variable, compile and test). Tool support for such operations is widely available, [4][7]. At a similar

level to these atomic refactorings are refactorings such as Extract Method which moves a section of (possibly repeated) code into its own method [5]. These refactorings involve only a few more operations than the atomic refactorings previously mentioned and are thus considered atomic. Tool support for these refactorings is also easily found.

3.1 Sequential Refactoring

The complexity of refactorings quickly increases with the combination and repetition of atomic refactorings. An example of this is Extract Class which encompasses Move Field and Move Method [5]. We term such refactorings sequential as they are sequences of atomic refactorings. Tool support for these refactorings is not as easily found.

The key point to address concerning atomic and sequential refactorings is that they maintain local knowledge only, of the application on which they function. An example is Rename Variable – the rename refactoring only needs to know the name of all other variables within the scope of the variable to be renamed. At most atomic and sequential refactorings need to know names of the classes in a given package, not the structure or interaction between the classes.

3.1 Complex Refactoring

We introduce complex refactorings as an extension of atomic and sequential refactorings and distinguish them from atomic and sequential refactorings in four ways. Firstly, complex refactorings are comprised of a series of atomic and sequential refactorings. In Section 5 we describe how tool support for atomic and sequential refactoring is easily integrated into the system. From a theoretical view, these small operations are easily incorporated into a larger operation. Secondly, complex refactorings have access to knowledge of the structure of the system. Using this information as a start point, complex refactorings know what classes to change. Thirdly, complex refactorings have access to knowledge about design patterns so that they know to what structure to change. Lastly, complex refactorings have the capability to generate code for classes required in a design pattern but unimplemented in the original system. The new classes are not simply a result of moving existing code to another location in the system, but are generated from scratch.

The point to be emphasized is we view complex refactorings differently than the original definition we provided for refactoring. While complex refactorings clean up code, they also require application and domain knowledge, perform broad application transformations and generate a determinable amount of code for each design pattern. Like atomic and sequential refactorings, however, they do not modify the behaviour of the system, from an end user point of view.

4 Tool Knowledge

Tool support to achieve complex refactoring to design patterns requires three knowledge stores. First, the tool maintains an *Initial System Store* which has knowledge of the original structure of the system and the user's design decisions. The Initial System Store asks the user what files need to be analyzed, what pattern is to be implemented, any nested patterns that should be implemented, names of new classes to create and various other design attributes of the original application. Through this series of questions the system establishes knowledge of the initial structure of the application and the requirements for the target application.

Secondly, the tool maintains a *Rule Store* where it stores rules and guidelines for the domain in which we are working. The complex refactorings we discuss work with J2EE type files. The Rule Store also maintains knowledge of .jsp files such as the extension, the Java code to scan, and any flags to ignore (e.g. HTML related code) as well as information about servlet and general java files. The Rule Store maintains information about the structure of session beans and entity beans. Finally the Rule Store maintains information about design patterns and the classes that are required to be implemented for each design pattern.

The final knowledge aspect maintained by the tool is the *Target System Store*, where all the complex refactorings reside. The Target System Store accesses the Initial System Store and the Rule Store to find out what needs to be changed and how it needs to be changed. At the time of writing the steps to create a given design pattern are represented as a series of complex refactorings represented as workflow in the Target System Store.

There are numerous design decisions we made when initially working on the tool. These design decisions concern the initial and target application, not the design of the tool, and thus these decisions must be made each time a developer uses the Design Pattern Developer. Firstly, what new structure should the target application have, if any? The user needs to specify if a 3-tiered structure should be implemented or if no new structure is needed at all. Secondly, what design pattern should be implemented and what patterns (if any) should be nested? There are also various pattern-specific decisions that a developer using the tool must make before applying change. Specific examples are provided in Section 5. Figure 1 gives an overview of the change the tool performs and a specific example provided in Section 5.

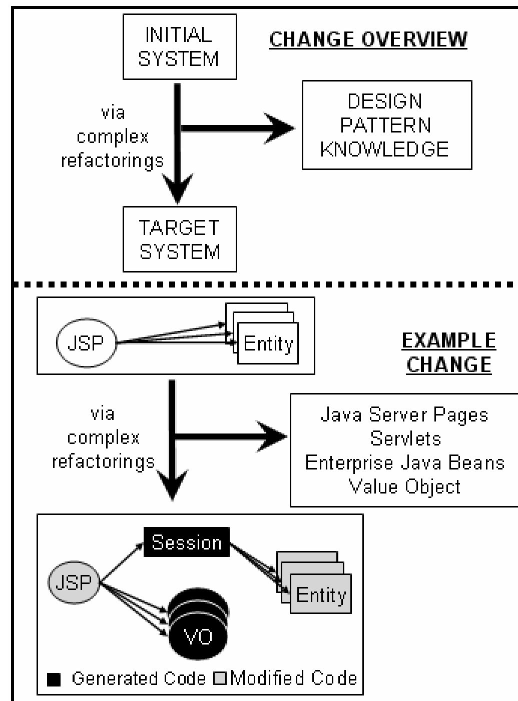

Fig. 1. Overview of the change occurring in the Design Pattern Developer; Example of Change in Session Façade with nested Value Object

5 Example

The Design Pattern Developer is implemented as a multi-page wizard, plug-in in Eclipse. The first three pages pose design questions to the user. The next four pages implement the actual changes based on information specified in the design decision pages. The Design Pattern Developer has been tested with an implementation of the J2EE business tier design patterns: Session Façade with a nested Value Object. After the desired pattern and 3-tier structure are decided, the following page specifies pattern-specific questions. For the Session Façade pattern we ask whether we need to be concerned with direct or sequential logic. We define 'direct logic' as a single line of code in the client file that needs to be moved to the Session Façade; we define sequential logic as multiple lines that must be moved to the Session Façade. We also ask for the ratio of session façades to entity beans. For purposes of application size or business logic organization, one may wish to implement more than one Session Façade for the entity beans in an application. The last design decision that must be made is if there are any patterns that should be nested inside the previously selected

pattern. In our example, we nest the Value Object pattern inside the Session Façade. Once the user has entered this information s/he can proceed to the rest of the wizard.

The fourth page of the Design Pattern Developer asks the user to select which .jsp, servlet and entity beans should be analyzed. If a new 3 tier structure was specified in the design decisions, it is implemented here. Next the name of the session bean to be created is specified (default to SessionFacade.java). If a 1:Many Session Façade to Entity Beans ratio design decision was made, the tool inserts an instantiation of the Session Façade in each of the client files and creates the actual session bean. If a Many:Many ratio was chosen the developer must then group the entity beans together and the tool creates one Session Façade for each group. An instantiation of the respective Session Façade is placed in each client file, depending on the entity bean the client file references.

The sixth page (Figure 2) defines the mapping between entity beans and value objects. At the time of writing the implementation creates one value object for every entity bean in the application. The value objects are created with all fields listed in the entity beans and accessor methods for each field. The final page lists and analyzes all client files that are changed. If direct logic was selected in the design decisions the following occurs: References to entity beans are changed to be references to respective value objects. Method calls on entity beans are changed to method calls on the session bean. If sequential logic was selected in the design decisions, the developer must select the text to be placed in the Session Façade and confirm the method name and content before it is placed in the Session Façade.

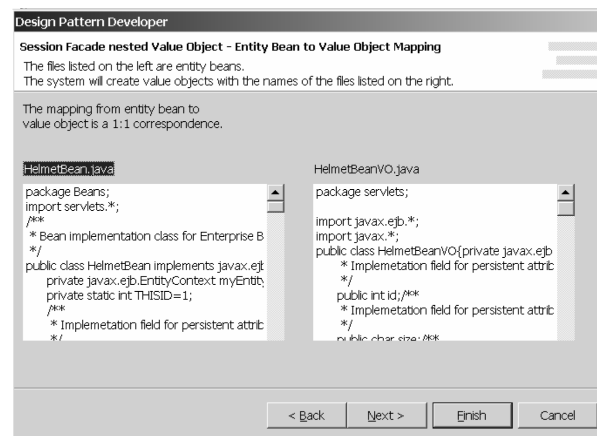

Fig. 2. Page 6 of the DPD. Entity beans to be modified listed on the left, corresponding value objects listed on the right. The code of each can be viewed.

6 Contributions

Tool support for refactoring is widely available ([4][7][10]). The concept of refactoring to patterns is not a novel one either ([6][7]). The novel contributions of this work, then is as follows:

i. We focus on integrating existing refactorings that have already been proven to be behaviour preserving.

ii. ii. We focus on J2EE applications, where we abstract the initial and target applications and place this abstraction in a separate package.

iii. iii. We access design pattern knowledge and specify the access of this information as being separate from the information that existing refactoring access.

iv. iv. We provide three definitions of refactoring based on the scope of the change that occurs and the information required to implement the change.

7 Current State and Future Work

To date we have implemented and are working on Session Façade, Value Object, Service Locator, Session Façade with nested Value Object and/or nested Service Locator [2]. The development environment, Eclipse, [4], was chosen it 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.

We have performed preliminary testing of all these patterns on a J2EE application and also tested Value Object on a larger J2EE application, M-ASE [9]. In the former situation the Design Pattern Developer created a session bean (and/or value objects depending on the pattern applied) with references to a single entity bean. In the specified .jsp files, the tool also created an instantiation of the Session Façade and changed all returned variable references to the entity bean to value object references. Method calls on the bean (e.g. getters) were changed to calls on the Session Façade and the necessary methods were implemented in the session bean. Finally the tool created a value object corresponding to the entity bean we specified.

Immediate future work includes testing practicality and proficiency of the tool with a group of junior developers. A proposed study is to allow a student group to manually change an application to match a design pattern and to change the application using the tool. Time required, errors introduced and overall comments on

the tool will be gathered. Run time of the target application will also be compared with the initial application.

8 Concluding Remarks

Traditional software development favours sound up-front design. Agile software development favours emerging design. We profit on the strengths of each approach by helping developers change the design of a existing software, regardless of whether the design was established up front or allowed to emerge throughout the development process. We propose complex refactorings that access knowledge of design patterns, design decisions, and the initial structure of the application as well as maintain the capability to generate code required to fulfill the requirements of a design pattern. The accompanying tool combines atomic and sequential refactorings to create these complex refactorings that access knowledge of an initial system, knowledge of design patterns, knowledge of how to change to a given design pattern and the capability to generate code. The domain is J2EE applications and preliminary tests have begun on two J2EE applications. The Design Pattern Developer can be extended to accommodate numerous complex design patterns, and assumes the user has knowledge of the motivation behind using each pattern. The desired goal is improvement in the following areas: readability, flexibility, understand-ability, development process and runtime.

References

1. Agile Alliance http://www.agilealliance.org/home (Last Visited: Dec 9, 2002).
2. 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.
3. Beck, K.; Extreme Programming Explained: Embrace Change; Addison Wesley Upper Saddle River NJ, 2000; p.103-115.
4. Eclipse www.eclipse.org (Last Visited: November 15, 2002).
5. 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.
6. Gamma E. Helm R. Johnson R. Vlissides J; Design Patterns – Elements of Reusable Object Oriented Software; Addison Wesley; Reading MA, 1995; p.1-3
7. IntelliJ IDEA www.intellij.com/idea (Last Visited: November 15, 2002).
8. Kerievsky, Joshua; Refactoring to Patterns; Industrial Logic www.industriallogic.com/papers/rtp015.pdf (Last Visited: November 15, 2002).
9. M-ASE http://sern.ucalgary.ca/~milos/ (Last Visited: November 15, 2002)
10. Roberts, D., Brant, J., Johnson, R.; A Refactoring Tool for Smalltalk; Department of Computer Science, University of Illinois at Urbana-Champaign