UNIVERSITY OF CALGARY

Tool Support for Complex Refactoring to Design patterns

by

Carmen Zannier

A THESIS

SUBMITTED TO THE FACUTLY OF GRADUATE STUDIES

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTERS OF SCIENCE WITH A SPECIALIZATION IN

SOFTWARE ENGINEERING

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

August, 2003

UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled “Tool Support for Complex Refactoring to Design Patterns” submitted by Carmen Zannier in partial fulfillment of the requirements for the degree of Masters of Science with a Specialization in Software Engineering.

Supervisor, Frank Maurer, Department of Computer Science

Guenther Ruhe, Department of Computer Science

External Examiner, Gail Kopp, Faculty of Education, Division of Teacher Preparation

Date

ABSTRACT

Using design patterns is seen to improve the maintainability of software systems. Applying design patterns to an application often implies upfront design often found in traditional approaches to software development. Agile approaches to software development apply an emerging approach to design where the design of a software application changes continuously with added requirements, an approach not suited to the use of design patterns. The gap between agile and traditional development is bridged via complex refactoring towards design patterns. Complex refactorings incorporate design pattern knowledge to introduce large changes to an application that result in design patterns. Existing refactorings are categorized into one of three groups: atomic, sequential or complex, based on four criteria. These new categories organize refactorings by their potential for change and by the knowledge they require to introduce their change. Complex refactorings are presented and are based on existing tool-supported refactorings, knowledge of the application to be changed, knowledge of design patterns, and the capability to generate code. A proof of concept of tool support for complex refactoring to design patterns is detailed and empirical results in favour of such a tool are given.

ACKNOWLEDGMENTS

Grateful acknowledgment is made to my supervisor Frank Maurer for his guidance and teachings throughout the development of this work. Many thanks also to members of the Computer Science department at the University of Calgary, especially those in the Software Engineering lab and specifically my research group, for their support and assistance.

DEDICATION

For Mom, Dad and Lia.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

The task of developing software can be approached using numerous methodologies and combinations therein but one line that can be drawn is between agile software methodologies and traditional software methodologies. Agile software methodologies encompass methodologies or practices such as SCRUM [Schwaber02], Extreme Programming [Beck00], and Crystal [Cockburn03]. Traditional software methodologies encompass methodologies and models such as waterfall [vanVillet93, 32], spiral [Boehm85], and incremental [vanVillet93, 38]. A line can be drawn between these two methodologies due to conflicts in the approaches to design. The conflicts between aspects of design in each methodology motivates this research.

1.1 Agile vs. Traditional Software Development

Agile methodologies rely on the YAGNI (You Ain't Gonna Need It) principle [YAGNI03]. Agile methodologies assume that creating more flexibility than is currently needed often is wasted effort. The belief is that a developer cannot predict future requirements or future changes to the system being developed so s/he should code for only what s/he *knows* is required. Design is created for what is needed now, and a more complex design is allowed to emerge as functionality is added to the system.

In comparison, traditional methodologies implement upfront design and attempt to include flexibility in their applications that is thought necessary or useful at the time of development for future changes to the system. The belief is that a developer can provide implementation for future requirements or changes *while* providing implementation for the current requirements. Design is created for what is needed now and what may be needed in the future. The YAGNI principle versus provided flexibility is a key difference between agile and traditional software development methodologies.

Design patterns enhance the readability, maintainability and flexibility of a software application, [GoF95]. Design patterns can be more easily used with software development methodologies that implement upfront design such as is performed in *traditional* software development methodologies. To bridge the gap between provided flexibility via design patterns and the YAGNI principle used in Agile methodologies, applications developed via agile methodologies must be allowed to change towards a design pattern. Agile methodologies must be able to integrate a design pattern after the initial design is coded. The research question addressed therefore is *can one have emerging design to design patterns*?

1.2 Research Motivation

By pursuing this research question, if successful, one is able to reap the benefits of both design patterns and agile methodologies. By emerging towards design patterns one is able to follow the YAGNI principle and produce a design that enhances readability, maintainability and flexibility [GoF95].

1.3 Goals

The specific goals of this thesis fall under the umbrella of providing support for emerging design to design patterns. This work thoroughly discusses Refactoring to Design Patterns to support design emerging towards design patterns. The goals are as follows:

- To define refactoring on three different levels of complexity and knowledge. This is done to better organize refactorings, estimate the time required to apply a refactoring and lastly to specify a category of refactoring used in refactoring to design patterns.
- To introduce and thoroughly describe complex refactoring to design patterns. The complex refactorings introduced encompass design pattern knowledge, initial application design knowledge and the capability to generate code.
- To present requirements for tool support to refactor to design patterns, a tool that helps agile software developers change the design of existing software to conform to a design pattern more typically found in top-down development methodologies.
- To present a proof of concept of tool support for refactoring to design patterns.
- To provide results of experimentation with the tool.

1.4 Hypotheses

The hypothesis to the question: "Can one have emerging design to design patterns?" is Yes. *Firstly*, it is thought that the existing agile practice of performing continual small refactorings on an application during development can be combined and grouped together to refactor to design patterns. *Secondly*, it is thought that existing tool support for performing said continual small refactorings on an application during development can be combined and extended to produce tool support for larger refactorings, specifically refactorings to design patterns. *Finally*, it is thought that applying complex refactorings and tool support therein will assist in automating (although not fully automate) the task of transforming an application absent of design patterns, to an application adhering to a given design pattern. The combination of these three hypotheses and the completion of each is support in favour of achieving design emerging to design patterns.

1.5 Expected Results

In defining three levels of refactoring it is expected that most existing refactorings (such as those found in [Fowler00]) will fall into the lower complexity categories (atomic and sequential). In terms of implementation, it is expected that the key requirement for tool support will be complex refactorings that require user input to accomplish their task. Lastly, it is expected that use of the tool to apply design patterns will be faster than manually transforming an application – particularly interesting when one considers potential logic and syntax errors introduced into the application.

1.6 Thesis Overview

The remainder of this work is as follows: Chapter 2 gives background information concerning refactoring definitions, properties and available tool support and discusses some domain knowledge pertinent to this research. Chapter 3 gives an overview of the research process. Chapter 4 explains four criteria upon which a refactoring is analyzed, gives justification for these criteria, establishes definitions for three new categorizations of refactoring, and gives examples of these three. Chapter 5 explains the proof of concept tool developed in eclipse. Chapter 6 discusses the experiment conducted using the tool. Chapter 7 concludes this work.

2 LITERATURE REVIEW

The following chapter gives background information in refactoring and design patterns. The chapter begins with a look at various definitions of refactoring, why and when refactoring occurs as well as some issues surrounding refactoring. The refactoring section is concluded with a look at various groups or categorizations of refactoring. The issue of refactoring to patterns is given in section two. A historical overview of refactoring browsers and two tools used in conjunction with refactoring are given in section three, and an introduction to design patterns is given in section four. Information concerning Java 2 Enterprise Edition applications, the application domain this research works in, can be found in Appendix A. The chapter concludes with a chapter summary.

2.1 Refactoring

The simple definition of refactoring is "cleaning up code," [Fowler00, xvi], but the use of refactoring, and the implementation of various refactoring browsers has taken that definition much further. This literature review begins with a look at the various definitions provided for refactoring and some concerns that go along with the process of refactoring.

2.1.1 What is Refactoring?

Refactoring as a noun is, "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour" [Fowler00, xvi]. As a verb, to refactor is, "to restructure software by applying a series of refactorings without changing its observable behaviour" [Fowler00, xvi]. Software restructuring is defined in [Opdyke92, 7] as, "the modification of software to make the software 1) easier to understand and to change or 2) less susceptible to error when future changes are made." Similar to the first definition given, this could be a definition for refactoring. In [Roberts99, 37] a more formal definition of refactoring is provided as a program transformation that has a precondition and a post-condition that a program must satisfy for the refactoring to be easily applied. More formally, "a refactoring is a triple R=(pre, T, P) where pre is an assertion that must be true on a program for R to be legal, T is the program transformation, and P is a function from assertions to assertions that transforms legal assertions whenever T transforms programs" [Roberts99, 37]. To be clear, here the post-conditions, P, are not just a statement of assertions, but a transformation between assertions. Assertions can be represented by one of two types of analysis, to determine if a program meets criteria to perform a refactoring. Primitive analysis is "the fundamental functions that a program must compute to implement the refactoring" [Roberts99, 30]. Derived analysis represents the preconditions of refactorings that can be computed from primitive analysis functions. Examples can be found in [Roberts99, 30], but in essence, the derived analysis functions are syntactical breakdowns of what the primitive analysis functions state. For example, IsClass(*class*) is true if a class named *class* exists in the system, and is false otherwise. This is a primitive analysis function for a refactoring such as AddClass. The derived analysis function is Subclasses(class) = {c | Superclass(c) = class}. [Roberts99, 31].

Informally, refactoring is any change that tries to improve code. Improving code is recognized at this point, as being a subjective decision, one the developer makes during programming. To what degree code is improved is a qualitative measure and can be based on qualities such as enhanced readability, comprehension or clarity, for example.

2.1.2 Why Refactor?

According to [Fowler00, 55], one refactors for four main reasons. Firstly, refactoring improves the design of existing software. As time goes by and code is changed, the structure or design of software crumbles [Fowler00, xvi]. By refactoring one slows this decaying process, and even halts it. Also, a design lacking in quality often is a breeding zone for specific problems such as duplicated code. By refactoring away poor code one improves the design of the application. The second purpose for refactoring is to make the software easier to understand. Whether reading someone else's code or reading one's own, one can refactor to better represent what the system is actually doing. By changing the code to explain exactly what it does, refactoring brings the bugs into the open. Lastly, refactoring emphasizes good design which speeds up the development process [Fowler00, 57]. Again, the concepts of improving design and making code easier to understand, at this point, are subjective decisions left to the developer. Neither a specific refactoring, nor this research, delves into what degree design or code is improved upon by applying a refactoring.

2.1.3 When To Refactor?

There is typically no set time to refactor; it should be done continually, whenever a section of code requires some tidying up. Some common rules of thumb are as follows: refactor when adding functionality, to improve code comprehension or to rearrange the code affected by new functionality. Secondly, refactor when you need to fix a bug and lastly, refactor when code is reviewed [Fowler00, 57].

2.1.4 Refactoring Issues

[Opdyke92] addresses the issue of refactoring being a behaviour preserving operation. According to [Opdyke92, 27], the following seven properties must be fulfilled to define a refactoring as being behaviour preserving. Firstly, all classes must have at most one unique superclass. All classes must have distinct class names and distinct member names. Inherited member variables must not be redefined. Signatures must be compatible in member function redefinition. That is, when a subclass is redefining a method found in a superclass, the signatures of the method must be the same. There must be type-safe assignments, meaning the values assigned to an attribute must be of the same type as the attribute itself. Lastly, there must be semantically equivalent references and operation. That is, when given a set of inputs to a program and generating a set of outputs from these inputs, after applying the refactoring and running the program with the same set of inputs, the outputs must be the same as the original set of outputs.

The idea of behaviour preserving is defined differently in [Roberts99, 60]. Each program is thought to have a specification for it and that specification is satisfied (or unsatisfied) by a test suite. A refactoring is therefore behaviour preserving if it satisfies the original test suite. If a new component is added to the program, the program must satisfy the original test suite plus any additional tests. [Roberts99, 63] When satisfying the original test suite, one must recognize that this is the conceptual original test suite that is satisfied. For example, if the refactoring Rename Method is used on any method called by a unit test, the refactoring must also update the test with the new name of the method. For example renaming a method from **loginA** to **loginAdministrator** conceptually does not change the application. A test that calls **loginA** must be changed to call **loginAdministrator** but conceptually the test is the same. Thus, when one says the original test suite must be satisfied, it is the concepts of the original test suite that remain unchanged. The syntax may however change to accommodate the refactoring.

There are two other refactoring issues that arise, the first of which occurs in applications that work with databases. If a business application is tightly coupled to a database schema, then refactoring code in the application may cause problems with the database-application relationship. Specifically, if changing names in code that directly relate to a database, table or column name, the association between the application and the database will not function. Secondly, refactoring interfaces is likely to cause problems. If one does not have access to all the code that implements an interface, a refactoring as simple as Rename Method, [Fowler00, 273], can cause problems. A developer may change the name of a method listed in an interface to which s/he does not have access. [Fowler00, 64] thus makes the important distinction between a public interface and a published interface, where a published interface is one that is used by multiple classes, to which a single developer may not have access and a public interface can be seen and changed by all developers. A refactoring that affects a public interface is okay, a refactoring that affects a published interface is not. Refactoring a published interface is extremely risky in terms of compilation errors alone.

In general, refactoring in and of itself is rather risky and has the potential to set back a programmer days and even weeks [Fowler00, xiii]. Consequently, the first step in refactoring is providing a solid set of test cases to run before and after the refactoring.

2.1.5 Refactoring Groupings

Martin Fowler provides numerous refactorings and describes them in terms of a name, a summary of what the refactoring does, the motivation behind the refactoring, the mechanics or actual steps in implementing the refactoring and lastly, an example. The refactorings are placed into specific groups depending on functionality. The <u>Composing Methods</u> group deals with problems related to methods and parameters. Duplicate code in a single class, lengthy methods, and long parameter lists are just some of the motivations behind refactorings such as *Extract Method*, *Inline Method*, and *Replace Parameter with Method* refactorings. The second refactoring group is titled <u>Moving Features Between Objects</u> and addresses where to place responsibilities in object design. Large classes, multi-purpose classes and feature envy are example motivations for using refactorings such as *Move Method* and *Extract Class*. The <u>Organizing Data</u> group includes refactorings such as *Replace Data Value with Object* and is motivated by the desire to simplify working with data. The fourth refactoring group is called Simplifying Conditional Expressions and deals solely with handling conditional expressions. Making Method Calls Simpler is motivated by long parameter lists and data clumps in objects. Refactoring solutions given are *Preserve Whole Object* and *Introduce Parameter Object*, respectively. Lastly, the Dealing with Generalization group provides refactorings such as *Pull Up Field*, *Pull Up Method* and *Push Down Field* and *Method*, respectively [Fowler00].

[Opdyke92], focuses on describing an approach for providing automated support to restructure object oriented frameworks and provides three high level refactorings. The three higher level refactorings are motivated by three concepts that Opdyke first establishes. First, Refactoring to Generalize motivates the refactoring of creating an abstract superclass. Refactoring to Specialize looks at subclassing and simplifying conditionals. Lastly, Refactoring to Capture Aggregations and Components looks at relationships among classes. Twenty-six low-level refactorings provided in [Opdyke92] focus on creating, deleting, changing and moving entities.

The refactorings from [Roberts99] are grouped into three categories, similar to that of the twenty-six found in [Opdyke92]. Firstly are class refactorings, such as addClass, renameClass, removeClass. Secondly are method refactorings (add, rename, remove, move) and thirdly are variable refactorings (addInstance, removeInstance, pullUp, pushDown).

2.1.6 Grouping Refactorings for Execution

When grouping and ordering refactorings for execution, from [Roberts99, 39], one cannot merely “AND” all preconditions of each refactoring together. The preconditions of a later refactoring may be satisfied by the post-conditions of an earlier refactoring. Instead, from the post-condition assertions, one can derive the grouped refactoring’s preconditions. There are two more points to be addressed from [Roberts99]. Firstly, when undo-ing a refactoring, [Roberts99, 56] points out that since a refactoring is behaviour preserving, the inverse of a refactoring must also be a refactoring and in a chain of refactorings, one can look at the dependencies between refactoring to determine which refactoring must be undone.

Much theory of refactoring has been developed and discussed and what is now becoming the focus is providing tool support to assist in applying the refactorings.

2.2 Refactoring to Patterns

The concept of refactoring to patterns is not a novel one. It is mentioned in [GoF95, 3] where design patterns are recognized as solutions to common software problems. In [Fowler00,], design patterns are seen as the targets for refactorings. "…Patterns are where you want to be, the refactoring are ways to get there from somewhere else." [Fowler00]. This idea is expanded upon in [Kerievsky03, 15] where a key point is made: design patterns are useful, in a certain context. One must be careful not to become design pattern happy, a situation where one uses design patterns just because s/he can. The design pattern may or may not be required, and is likely unnecessary in many situations where it is being used (e.g. using the Decorator pattern to create a HelloWorld program). In light of the increased popularity of agile methods, [Kerievsky03] recognizes the effectiveness of continual and changing design and the lack of necessity for some aspects of upfront design. "When you make your code more flexible or sophisticated than it needs to be, you over-engineer it." [Kerievsky03, 7] According to [Kerievsky03], agile methodologies help to avoid over and under-engineering. One of the key tenets of agile methodologies is continual refactoring, but design patterns should not be left for good. The better option, according to [Kerievsky03], is to refactor to patterns, when one finds a place where a pattern helps to improve design. [Kerievsky03] finds that the motivation for refactoring to design patterns is the same as the motivation for implementing non-patterns-based refactorings, i.e. to reduce or remove code duplication, to simplify code or to make code more understandable. "The motivation for refactoring to patterns is not the primary motivation for using patterns that is documented in the patterns literature." The motivations for refactoring to a design pattern found in [Kerievsky03, 10] are issues such as code duplication, communicating intent and reducing creation errors, whereas the motivation for using a design pattern initially, is often a design issue and this is clear in the examples provided in [Kerievsky03, 10]. The bulk of the work in progress by [Kerievsky03] is documentation discussing when to refactor to patterns as well as providing specific refactorings for these scenarios. The goal is to "see patterns in the context of refactoring, not just as reusable elements existing apart from the refactoring literature." [Kerievsky03]

[Tokuda99] discusses refactoring to design patterns from a slightly different angle: the difficulty of evolving legacy systems. Two points made are that the benefits of source to source transformations have not yet been quantified, and that in making the transformations to object oriented design the users should be allowed to name any new entities introduced to the system via the transformation. Examples of this are the prompts provided by Eclipse [Eclipse03] and IntelliJ [IntelliJ03] for a user to enter names of new classes or other attributes of the system.

2.3. Refactoring Browsers

While getting started with refactoring seems at first to be laborious, tools are becoming widely available. Experience reports that while refactoring initially seems time consuming, the upfront effort of refactoring far outweighs the cost of not refactoring at all, later in the development process [Fowler00]. With the increased popularity of refactoring, development environments with built in refactoring support have quickly become the norm. What follows is an overview of four refactoring browsers.

2.3.1 SmallTalk Refactoring Browser

The SmallTalk Refactoring browser was the first refactoring tool developed and is the basis for the structure of the refactoring support provided in eclipse [Roberts et al.99]. The browser bases all its refactorings on three principles. First, is that the refactorings can be automated, (although not completely automated) second is that they are provably correct and lastly is that more complex refactorings can be created by composing primitive refactorings [Roberts et al.99]. The Smalltalk Browser was motivated partially by the desire to refactor to improve program comprehension. By restructuring the code, one sees where the errors lie and one is able to better understand the system.

The browser follows four specific criteria [Roberts et al99]. Firstly, the refactorings are integrated into standard development tools, as occurs with this refactoring browser, IntelliJ [IntelliJ03] and Eclipse [Eclipse03]. Secondly, the refactorings must be fast, at least as fast and more than likely faster than it takes to manually refactor. Thirdly, the refactorings should not be purely automatic, that is, as in Eclipse [Eclipse03], IntelliJ [IntelliJ03] and the Smalltalk browser [Roberts et al.99], the refactoring should require at least some user interaction. Lastly, the refactorings must be reasonably correct to generate user trust of the tool. The SmallTalk refactoring browser assumes there is developer intelligence in executing the refactoring process.

In the implementation of the browser, the basis of the source code transformation is a parse tree to parse tree transformation, not a string to string transformation. Consequently, a tree matching system was developed in [Roberts et al.99]. Each refactoring also satisfies a certain number of preconditions before implementing the transformation. Finally, functionality to support undo via a change history was next in line to be worked on, at the time of writing of [Roberts et al.99].

2.3.2 Design Pattern Tool

The tool described in [Cinneide00] was developed to refactor to design patterns. The Design Pattern Tool is implemented as a four layer architecture of a)Design Pattern Transformations b)Minitransformations c)Helper Functions, predicates and refactorings and d)Abstract Syntax Tree operations. Based on [Opdyke92] and [Roberts et al.99], the tool developed by [Cinneide00] details behaviour preservation. It outlines mini-transformations as the method by which a pattern is introduced into an application.

At the time this research was performed, development environments such as Eclipse and IntelliJ were just getting started. While the work done in [Cinneide00] is closely related to this research, follow up work has not been provided since. Due to the timing of this work, a rather key weakness is its lack of incorporating existing refactorings. The tool contains mini-transformations conceptually similar to the sequential and complex refactorings introduced in this research, but lacking the integration of existing tool support. According to [Cinneide00] a refactory for Java was developed, with an extensive group of refactorings. This lack of integration also requires the tool to support an Abstract Syntax Tree layer with its refactorings. There is no follow up work to this research and thus its popularity and use of the tool is minimal.

2.3.3 Eclipse

Eclipse is an open source development environment by IBM [Eclipse03]. Eclipse supports numerous plugins, and the capability to develop and support self-made plugins. Various newsgroups exist for on-line help as well as numerous online manuals. Primarily a Java development environment, eclipse comes with plugins to handle CVS [CVS03], Ant

[Ant03] and JUnit [JUnit03], just to name a few. Its refactoring support is ever-growing and the refactoring class structure is based on that of [Roberts et al.99].

Eclipse was chosen as the development environment for this research due to its availability of code, plug-in support and refactoring support. Eclipse contains refactoring wizards, refactoring classes and change classes to handle each individual refactoring. When a refactoring is called, the specific refactoring wizard is launched and user input is taken. When given the "go-ahead" from the user to perform the refactoring (i.e. 'OK' button clicked), preconditions are first checked. If the preconditions pass, an instance of a Change class is created and a change operation is performed. A ChangeOperation class performs the actual modification at the source code level. After the change operation, post-conditions are checked. All three of these steps: preconditions, change operations and post-conditions are encompassed in a specific refactoring class, a subclass of a generic refactoring class that exists in every refactoring wizard. The refactoring wizard is launched from any one of the available refactoring menus. [Eclipse03] A partial class diagram is provided in Appendix A.

While Eclipse's refactoring support is not yet as extensive as that of browsers such as IntelliJ [IntelliJ03], it is ever-growing and open to all developers. Learning the class structure and hooking into provided functionality is a viable option. One admirable aspect of Eclipse's refactoring support (and for that matter, CVS support as well) is the Preview feature. A developer can look at the change that is about to occur at the source code level and can then make a decision as to whether or not to implement the refactoring. In [Tokuda99], it is stated that a mature refactoring should be treated as a trusted tool. [Tokuda99] compares this trust to that of the trust a developer has for a compiler. While it may take some time to reach this level with refactoring, Eclipse has provided a means by which the developer can see exactly what is occurring right before it occurs. Consequently, the trust issue is somewhat alleviated.

2.3.4 IntelliJ

IntelliJ, created by JetBrains [IntelliJ03], advertises itself as the industry leading Java integrated development environment. It boasts features such as new refactoring support, intelligent code assistance, Java development features for rapid web application, code inspection tools, integrated versioning system and an open application programming interface for third party plug-in support. It has three types of code-completion, 100% of its features accessible by mouse and a friendly user interface. While the refactoring source could not be viewed, a trial version of IntelliJ is available for download and some experimentation was performed with this version.

The refactoring support provided by IntelliJ is extensive. Low-level refactorings such as rename variable or method are of course supported, as are more complicated refactorings such as Extract Interface, Replace Inheritance with Delegation, Encapsulate Fields, and Replace Constructor with Factory Methods. An apparent limitation of IntelliJ however, is its somewhat blind refactoring capabilities. When extracting an interface or extracting a superclass, one is not provided with any options (e.g. what package to place the class, imports to include, etc). The package in which the selection occurred when the refactoring was activated is where the class/interface is created. The developer only enters the name of the new class. Secondly, IntelliJ does not provide a strong refactoring preview. A small box appears showing the code of the new refactoring and the number of changes that will occur, but it is not as intuitive as the preview support provided in eclipse. The developer is left relying on somewhat blind trust that the refactoring will be performed correctly. Nevertheless, IntelliJ does perform the refactoring correctly, and makes updates through the entire source, respective to the refactoring.

2.3.5 Supporting Refactoring Tools

The following tools are not refactoring browsers, but are advertised as providing support for refactoring. They both require a refactoring browser to be incorporated and are included here for completeness.

2.3.5a Elbereth

On a smaller scale, Elbereth is a Java implementation of a tool to support refactoring Java programs. [Korman98] recognizes the difficulty in identifying the structural problems with a current application’s design and finding the potential solutions. Elbereth approaches the problem of code duplication using a star diagram, which is, in reality, a tree structure of data structures in the source code. The data structures may be a given class, a primitive type or some other identifier. The root of the star diagram is a node which represents the identifier (variable, method, etc.) being analyzed. The number of references to the node is shown next to the name of the node. An interesting point is that the paths created by the tree represent possible interfaces for the system. In comparing star diagrams for two classes, the interfaces can be examined to identify similarities in code. This provides potential to refactor out duplicated code (for example into an abstract class) [Korman98]. Elbereth helps determine where the refactorings should occur provided it was integrated with an automated restructuring tool. It is not currently integrated with a refactoring tool.

2.3.5b DUPLOC

On a similar scale to Elbereth, DUPLOC is a tool to support refactoring duplicated code in the context of object oriented design. Its main features are a code reader, adaptable to different programming languages and a line based string comparison matching. DUPLOC provides a visual representation of duplications found in an application as well as textual reports identifying the duplicated sequences. Two topics are addressed in [Ducasse et al.99]: 1) refactoring operations on duplicated code and 2) refactoring duplication in an object oriented context. DUPLOC compares source objects, but does not divide source code into programming language entities (as Eclipse does). It compares two source objects and generates a report that shows the participants of the comparison, and a summary of matching sequences between the participants. Note that here again DUPLOC does not perform the actual refactorings and requires a tool that performs the actual refactorings.

2.4 Design Patterns

The major publication of design patterns was [GoF95], a detailed catalogue of numerous design patterns and when they should be applied. A design pattern consists of a name, the problem that motivates the pattern, the solution to the problem, often including an example with a class diagram, and finally the consequences of applying the pattern. The design patterns in [GoF95] are, “descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context” [GoF95, 3]. The patterns solve design problems such as finding appropriate objects, determining object granularity, specifying object interfaces and implementations and the like. The book, however, talks about designing for change by, “anticipating new requirements and changes to existing requirements…so that [the system] can evolve accordingly." As previously mentioned, this is not the only, and from an agile perspective likely not the best reason to use a design pattern. More useful in terms of agile methods are the steps provided in how to select a design pattern. The original catalogue of design patterns found in [GoF95] consists of twenty-three design patterns grouped into three categories: creational, structural, and behavioural. Creational design patterns are concerned with separating a system from how the objects in it are created and represented. Structural design patterns deal with the composition of classes to form larger structures. Behavioural design patterns are concerned with allocating responsibility to objects. In [GoF95] the design patterns can be used for various types of applications, but were originally made for a standard software application, not necessarily related to the internet. In [Alur et al.01] there are numerous patterns specific to Java 2 Enterprise Edition applications. [Grand02] also provides numerous design patterns for distributed and enterprise applications.

For the purposes of this research four patterns were used as examples, one pattern found in [GoF95, 127] and three found in [Alur et al.01]. These are described briefly.

Singleton: The singleton design pattern is a creational design pattern that establishes a class such that there is only one instance of it, and all objects have access to that instance.

Value Object: The value object design pattern is a business tier J2EE design pattern. It establishes data classes in relation to entity beans. To avoid numerous network calls to entity bean accessor methods, the entity data is placed into one object which is passed across the network. Any individual read accessor calls to the value object occur locally.

Service Locator: The service locator design pattern is a business tier J2EE design pattern. It is somewhat like a singleton, in that a class is created with only one instance of it. This class, however, incorporates all JNDI (Java Naming and Directory Interface [J2EE00]) lookup information and provides remote entities so that the lookup is done only once and the home interface need be retrieved only once for the entity beans.

Session Façade: The session façade design pattern is a business tier J2EE design pattern. It hides entity bean functionality from the client tier by maintaining all business logic and providing methods for the client tier that are representative of business workflow.

Note that while there are only four patterns listed here, they can be grouped in numerous ways to provide more combinations. Examples of this are Session Façade with nested Value Object and/or Service Locator. In chapter five we discuss more of this combining when explaining the proof of concept.

2.5 Chapter Summary

This chapter covered various definitions of refactoring, from a casual “cleaning up code” definition to those more formal. Issues surrounding refactoring were identified, not the least of which was that a refactoring must be behaviour preserving. Existing grouping or categorizations of refactoring were provided to better organize the purposes of refactorings. The concept of refactoring to design patterns was discussed and a historical overview of refactoring browsers was given. The chapter concluded with a look at design patterns. With this existing knowledge base covered, the following chapter gives an overview of the research design process of this work. The following three chapters delve into more detail of the concepts identified in chapter 3.

3 RESEARCH PROCESS OVERVIEW

Before delving into the details of what this research produces, there are definitions and assumptions that must be laid out. Throughout the remainder of this work terms related to refactoring are used in a specific manner that requires definition so as not to confuse the reader. There are also certain assumptions made that must be qualified. The first section of this chapter covers necessary definitions and the second section covers assumptions made. The third section then introduces the research design looking at what this research contributes. The chapter concludes with a summary.

3.1 Definitions

There are four definitions related to refactoring that appear throughout this text. They are as follows:

Pre-Refactoring Code – the code before a refactoring is applied.

Post-Refactoring Code – the code after a refactoring has been applied.

As a refactoring can be of varying complexities (to be discussed in chapter 4) the pre and post refactoring code of small refactorings can reside inside the larger refactorings.

Implementation of a Refactoring – the coding of tool support for a refactoring that is executable and used by a developer

Application of a Refactoring – performing a refactoring on an existing code base, either manually or with a tool

In terms of programming and establishing categorizations of refactoring (as found in chapter 4), two definitions must be established.

Programmable Entity – an element in code that creates a node in an Abstract Syntax Tree [Aho et al98]. For purposes of this research it can be one of field, method, class, package, project

Scope of Programmable Entity – the level of the programmable entity. Field is the lowest scope, then method, then class then package then project. The scope of a programmable entity can be seen as the potential for a programmable entity to encompass other programmable entities.

3.2 Assumptions

One key assumption is made in this research. The assumption made is that when categorizing refactorings, as done in chapter 4, one does not consider references to refactorings as affecting the category of the refactoring. Numerous existing refactorings propagate changes throughout the rest of the application, even if the refactoring is as small as renaming a variable. The categorizations are based upon the initial change that is made by the refactoring. This is done because the changes propagated beyond the initial refactoring are inevitable and can be performed automatically, but are performed to compile, not necessarily for the same reason the refactoring was performed. The categorizations of refactorings provided in chapter four are done to better understand the size of the change that occurs as a result of the refactoring. Specifying that changes occur throughout the program is redundant and does not help in understanding the complexity of the actual refactoring.

3.3. Research Design Process

This research examines bridging the gap between traditional approaches to design and design patterns, and agile approaches to design and design patterns. Thus one must look at the current state of the agile world, the chosen domain of this research. Numerous refactorings exist and tool support for some of these refactorings are available, as has been discussed in chapter two. The concept of refactoring is a common one to agile developers, specifically in Extreme Programming [Beck00] but the limits of the current state of refactoring are addressed in chapter three. This research categorizes numerous refactorings and expands on corresponding tool support. Figure 3.3.1 shows the research design process used for this work.

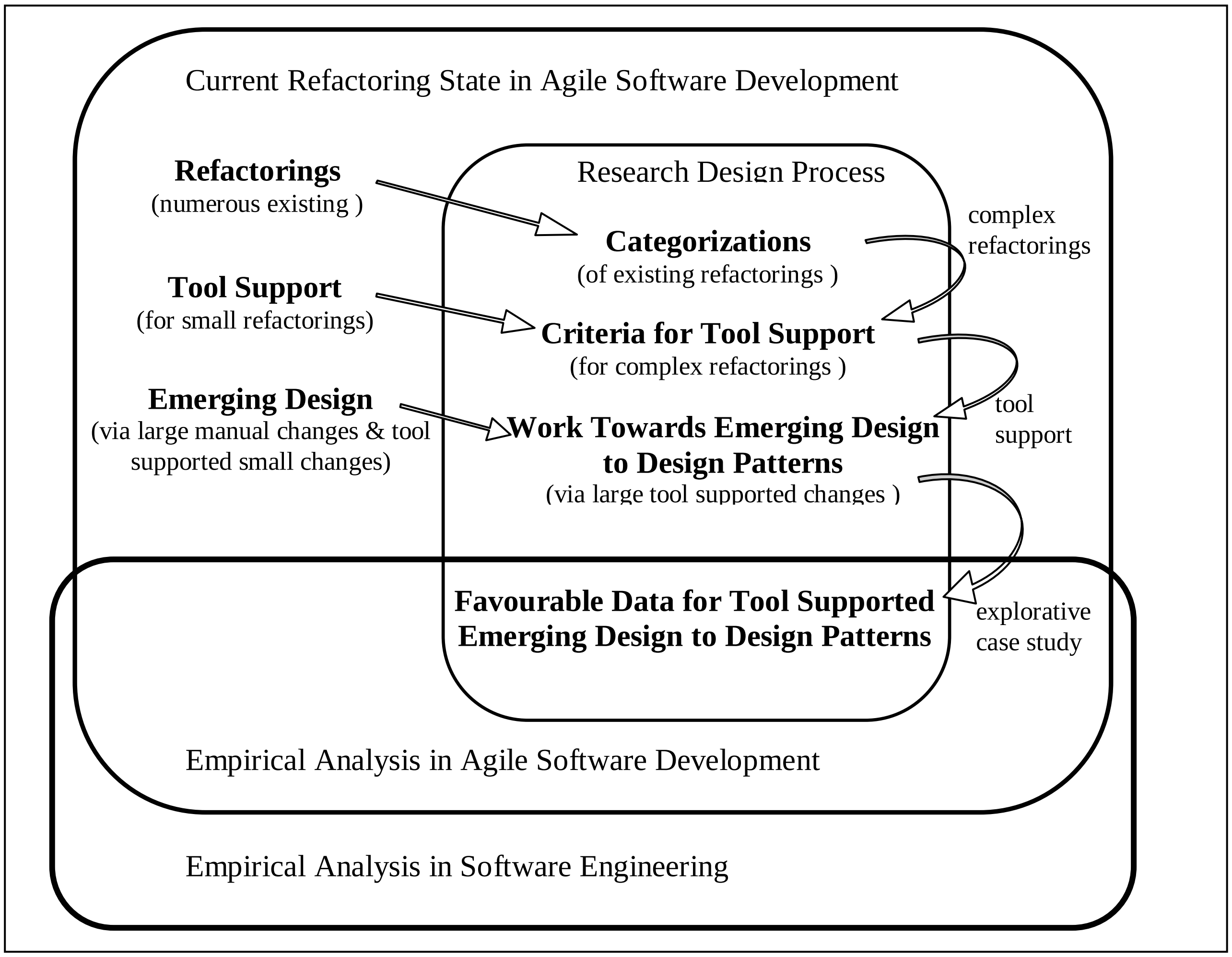

Current Refactoring State in Agile Software Development
Refactorings
(numerous existing)
Tool Support
(for small refactorings)
Emerging Design
(via large manual changes & tool supported small changes)
Research Design Process
Categorizations
(of existing refactorings)
complex refactorings
Criteria for Tool Support
(for complex refactorings)
tool support
Work Towards Emerging Design to Design Patterns
(via large tool supported changes)
Favourable Data for Tool Supported Emerging Design to Design Patterns
explorative case study
Empirical Analysis in Agile Software Development
Empirical Analysis in Software Engineering

Figure 3.3.1 Research Design Process

This research began with a look at the current state of refactoring and tool support for refactoring. Over 100 documented refactorings were examined and categorized into three distinct sections. Existing tool support was examined and built upon to support the top-level category of refactorings established. Meanwhile the concept of emerging design identified the approach Agile methods take to design and clarified the need for support for such an approach. In establishing too support for the top-level category of refactorings (complex refactorings) support for emerging design was introduced. To support or refute the work done for emerging design, an explorative case study was performed which provided initial results for tool support to refactor to design patterns, a step towards emerging design. This explorative case study exists in the realm of software engineering empirical analysis and more specifically in the realm of empirical analysis in agile software development.

3.4. Chapter Summary

This chapter has laid out some fundamental concepts used throughout this document. The confusion of refactoring as a verb versus a noun has been addressed as having various definitions related to refactoring. Two assumptions of this work have been established, the first related to the use of the term Abstract Syntax Tree and the second related to the issue of change propagation in refactoring categorizations. Finally, the research design process has been established in Figure 3.3.1. The following chapters detail refactoring categorizations, tool support and a study done to support this research.

4 REFACTORING CLASSIFICATIONS

Refactoring is a fix. It is a key tenet of Extreme Programming [Beck00] used to tidy up and organize code. If a programmer codes a method that requires numerous lines of commentary to explain what the method is doing, the code should be fixed instead. Accessing a field with accessor methods, rather than just accessing the field directly, is also a programming "rule of thumb" for many developers. These and other examples of refactoring should be performed on a regular basis, according to Extreme Programming [Beck00].

Refactoring is a change. The reason refactoring fits so nicely into Extreme Programming's mantra, is because refactoring is making changes to code, and Extreme Programming is about handling change. "The code does not support that concept." Well, make it. That is, change it, modify it, adjust it, do what needs to be done to satisfy the need for change.

Refactoring is a "do-over". A second chance to make a good impression, refactoring gives developers a chance to do a better job, while maintaining the end user functionality of the code base.

Regardless of what it is called: cleaning up code, improving readability, enhancing usability, fixing, changing, doing over, the concept of refactoring is a common-sense recognition that developers can continually improve. An issue taken with refactoring's popularity is an overuse, and sometimes misuse of the term. Seemingly any change to code can be qualified as a refactoring if it remotely cleans up code. On a conversational level, this is okay. On a practical level, this is not. It is the specific noun – a refactoring, that introduces issue into the practice of refactoring. What is provided in this chapter is a set of categorizations to better organize refactorings and to qualify refactorings on different levels of complexity for the purposes of implementing tool support to refactor to design patterns. Concretely categorizing refactorings has four immediate benefits. What follows is a discussion of the benefits of refactoring categories, the criteria on which the refactoring categories were defined, followed by the classifications of refactorings and examples. The chapter ends with a summary of what was discussed.

4.1 Benefits of Refactoring Definitions

4.1.1 Organization

Martin Fowler's initial book on refactoring, [Fowler00], is comprised of over seventy refactorings. Joshua Kereivsky's work in progress, [Kerievsky03] contains thirty refactorings, at the time of this writing. Alur, Crup et al. [Alur et al.01] have fourteen refactorings. These are already published refactorings, not to mention the numerous refactorings being used but yet unpublished. Some refactorings such as Move Method or Move Field stand on their own, some are the complete antithesis of another refactoring (e.g. Extract Method vs. Inline Method), and many are joined with others to create new refactorings (Change Value to Reference, Duplicate Observed Data, Encapsulate Collection, Replace Type Code with Class, [Fowler00]). Grouping refactorings and defining them based on specific criteria helps gain control over the numerous existing refactorings and combinations therein.

4.1.2 Understanding

In light of the growing numbers of refactorings available, the ability to keep up with the possibilities becomes quite time consuming. What likely happens is a developer becomes comfortable with the refactorings provided by his/her development environment, a limited supply at best. Even if one takes the time to read the refactoring catalogues available, one can easily get lost in the options, and come away with only a limited understanding of the complexity of each refactoring. Grouping and defining refactorings on specific criteria helps one understand the magnitude of each refactoring.

4.1.3 Task Estimations

Numerous refactorings such as those found in [Kerievsky03] or at the end of [Fowler00] require manual refactoring, which is incorporated in a developers' programming task and task estimate. By classifying refactorings by complexity, one specifies what is involved in each refactoring and, if no tool is provided for said refactoring, one is able to better understand the task before him/her and therefore better able to estimate the time required to complete said task.

4.1.4 Implementation

The refactorings listed in the catalogues [Fowler00, Kerievsky03, Alur et al.01] are clear and helpful. They provide specific examples and/or class diagrams the reader can follow (with some effort) to catch the general idea of the refactoring. When it comes to implementing a refactoring such that the process is partially automated, understanding is a new challenge. Implemented refactorings change the tree structure of a program and thus require access to the Abstract Syntax Tree. Implementation of more complex refactorings, as will be shown, requires access to more information than the Abstract Syntax Tree. Suddenly, the implementation of a refactoring such as Extract Composite, [Kerievsky03], becomes quite a task. Such refactorings require more information than smaller refactorings in order to successfully implement the refactoring. Implementing a refactoring requires encapsulating code change knowledge a developer has, into a code base. Defining refactorings based on specific criteria facilitates implementation by specifying the requirements for a refactoring.

The benefits of classifying refactorings stated, what is now presented are the criteria upon which the refactorings are categorized. Each of the 100+ refactorings from [Fowler00, Kerievsky03, Alur et al.01] were examined and commonalities were found among the refactorings. These commonalities are the basis for the refactoring categorizations and are described below.

4.2 Criteria to Analyze Refactorings

Each refactoring is a change in code and consequently, a developer has two pictures of his/her code: the pre-refactoring code, and the post-refactoring code. For the remainder of this thesis, the "pre-refactoring code" may also be referred to as the original source and the "post-refactoring code" may also be referred to as the target source.

4.2.1 Nested Refactorings

Definition - Nested Refactoring: If R is a refactoring and R' is a refactoring, R' is nested inside R if a step in R calls R'. R is said to have a nested refactoring.

The last point to be discussed concerning a refactoring is whether or not the refactoring has other refactorings included as part of its mechanics. While some of the refactorings provided in [Fowler00] operate on their own, numerous refactorings include one or more refactoring(s) to accomplish their task. A prime example of this is Collapse Hierarchy [Fowler00, 344] which uses (possibly numerous versions and iterations of) Push Down Field, Push Down Method, Pull Up Field and Pull Up Method [Fowler00]. In a case such as this, there is little reason not to use existing refactorings (why re-invent the wheel?). In other refactorings though, the presence of a nested refactoring is situational. An example of this is Move Field [Fowler00, 146]. As will be shown, a refactoring is defined partly based upon the type of refactorings (if any) nested inside of it.

4.2.2 Original Source Scope

Definition - Original Source Scope: The number and the type of one programmable entity surrounding the type of the programmable entity(ies) to be refactored.

The Original Source Scope is the scope of the code that will be refactored. The original source is the code to be changed. In some circumstances, the location is even where one clicks the mouse down in an editor and says, “There is a problem here.” It is the code section that is a result of the reason for the refactoring. It is the **result of** the reason for the refactoring, because the reason for refactoring is typically a conceptual one. The code is the result of the concept that needs to be changed.

The scope of the original source is the number and type of the next larger programmable entity surrounding the original source. The code to be changed may be encompassed inside a class, method, constructor, package, or even project. Since it is the next largest programmable entity *surrounding* the code to be changed, the Original Source Scope can never be a field of a class, because a field is the smallest programmable entity and does not surround itself.

Figure 4.2.1 shows the Extract Method refactoring example taken from [Fowler00]. One can click and highlight the lines immediately following the comment “print details” found in the first block of code and say “this is a problem.” The term “problem” is used loosely here – it is an aspect of code to be changed. This code is encompassed inside a method. Thus, the Original Source Scope is 1 method: the number + the type of the next largest programmable entity surrounding the original source.

```
void printOwing(double amount){
     printBanner();
    //print details
     System.out.println("name:" + _name);
     System.out.println("amount:" + amount);
}
```

```
void printOwing(double amount){
    printBanner();
    printDetails(amount);
}

void printDetails(double amount){
    System.out.println("name:" + _name);
     System.out.println("amount:" + amount);
}
```

Figure 4.2.1 Extract Method Refactoring Example [Fowler00, 110]

4.2.3 Target Source Scope

Definition - Target Source Scope: The number and type of the largest new programmable entity(ies) created by the refactoring.

The Target Source Scope is the scope of the code that has been refactored. The target source is the resulting code of a refactoring. The Target Source Scope is specified by two attributes. First is the number of the largest programmable entity created by applying a refactoring. Second is the type of the largest programmable entity created by applying a refactoring. In the Extract Method example the refactoring creates a new method. Thus, the Target Source Scope of the Extract Method refactoring is 1 method: the number + the type of the largest programmable entity created by applying a refactoring. Unlike the Original Source Scope, a field can be the entirety of the Target Source Scope. It can be all that is new after applying a refactoring.

4.2.4 Knowledge Type

Definition - Knowledge Type: The knowledge required to accomplish a refactoring, can be Abstract Syntax Tree knowledge, General Development Task knowledge and/or Design Pattern knowledge.

When implementing a refactoring, there is a certain amount of knowledge encompassed in the implementation. For example, when using the Extract Method refactoring an automated tool has to know how to create a method and place it in the file in which the refactoring was performed. Any refactoring tool has to have access to this information, or else the refactoring can not be implemented. This knowledge may be found in different classes and likely in different packages, and will differ from development environment to development environment, but conceptually the idea is consistent. The tasks to implement a refactoring are encompassed in one section of the implementation, the knowledge is encompassed elsewhere. Knowledge Type is the kind of information required to implement a refactoring.

When examining the numerous refactorings found in [Fowler00, Kerievsky03, Alur et al.01] one finds the type of information required for each refactoring differs in three ways. We define three levels of knowledge access and this is the third principle upon which a refactoring is categorized. The three levels are defined below.

4.2.4a Abstract Syntax Tree

Definition - Abstract Syntax Tree Knowledge: A change made to the Abstract Syntax Tree motivated by a programmer's subjective desire to restructure a section of code to enhance readability, comprehension, clarity.

An Abstract Syntax Tree is "a condensed form of a parse tree useful for representing language constructs," [Aho et al.98, 287] where a parse tree pictorially shows how the start symbol of a grammar derives a string in the language." [Aho et al.98, 29]. Thus, every time a section of code is changed the Abstract Syntax Tree for that file changes. When a refactoring changes a piece of code, it must propagate these changes to the Abstract Syntax Tree, and by default, it must have access to the Abstract Syntax Tree in order to make the change. The refactorings use the Abstract Syntax Tree to check that the refactoring can be implemented: that there are no syntax errors on the component being refactored. The refactorings also use the Abstract Syntax Tree to confirm that no syntax errors result from applying the refactoring.

4.2.4a Example

For example, in eclipse, when applying Extract Method to a java file, the ExtractMethodRefactoring class first checks that the refactoring can be activated. Activated generally means there are no syntax errors on the component, so the refactoring can begin. The activation method makes a call to the Abstract Syntax Tree. If all goes well (i.e. no syntax errors are found), the refactoring begins. Next, the ExtractMethodRefactoring checks the input in the ExtractMethodWizard where the user entered specifications for the refactoring. Here, the proposed change (new method name, new method contents) is sent along with the original Abstract Syntax Tree to see if the changes will work. Two checks are done: first to see if the new method is already used in the existing Abstract Syntax Tree, and second to see if the new method overrides a method already in the Abstract Syntax Tree or in a super class. If these tests pass, the refactoring is applied.

The access to the Abstract Syntax Tree is a minimal amount in comparison to the amount of information a refactoring could access. There are only a few (3) initial calls to the Abstract Syntax Tree, and for the purposes of preliminary checks only. Refactorings that require only this amount and type of information, (in Eclipse) apply the refactoring via change classes and the new file is re-parsed and re-compiled. Abstract Syntax Tree knowledge is the base level of knowledge a refactoring can have. At the very least, for compilation purposes, a refactoring will require access to the Abstract Syntax Tree, but can potentially require more. This is where the next two knowledge levels become advantageous.

4.2.4b General Development Tasks

Definition - General Development Tasks: The introduction or elimination of a programmable entity as part of day-to-day programming tasks, often accomplished through the use of tool support in integrated development environments.

General Development Tasks encompass the numerous tasks a developer performs when working on a software application: creating or deleting a package, class, method, or field. A tool that has to create (or delete) a new class has to know how to create (or delete) the class, the accepted structure, and naming convention. The same is true of a tool that creates (or deletes) a package, method or field. Most importantly, the same is true of any refactoring that performs any of these general coding practices. Thus, a refactoring that needs to create a component (or delete), needs to first access information at the Abstract Syntax Tree level, as well as information at the newly defined, General Development Tasks level. These General Development Tasks are typically encompassed inside the integrated development environment in which a programmer is working. If one is implementing tool support for refactoring in an integrated development environment, one likely already has access to the General Development tasks provided by that tool. One merely needs to include the required class files. It is important to note that these are not transformations on the Abstract Syntax Tree. General Development Tasks add or remove components, they do not solely modify existing components in an application thus they are additions or deletions, not transformations. While General Development Tasks may add to the Abstract Syntax Tree, they do so only as programming changes the Abstract Syntax Tree. Nevertheless, the idea of General Development Tasks and accessing them, is specified here for completeness.

4.2.4b Example

In eclipse, Java files are identified and accessed as ICompilationUnit. Classes can also be referred to as IJavaElements, as can packages, methods and fields. The compilation units can create import lines, packages, methods and fields directly or indirectly, and the compilation units are created via a CreateCompilationUnitOperation class. These classes, as they stand, have little to do with the refactoring support in eclipse, and are representative of the General Development Tasks developers perform. These classes, in fact, are used in eclipse to help developers create the various components they require. They encompass the knowledge that many refactorings can use in their implementation.

An example is the refactoring ExtractClass. An implementation of this refactoring would firstly need access to the Abstract Syntax Tree to confirm that the classes from which the refactoring will extract, will compile. Secondly, the refactoring would need access to the Abstract Syntax Tree to confirm that user input and proposed changes are valid for the refactoring. Next, in applying the refactoring, the tool would need to know how to create a class (i.e. the one it is extracting) and finally what fields to move. Moving involves knowing how to copy and paste and/or how to create in the target class then delete in the original (depending on one's implementation choices). These last steps: creating a class, copying and pasting methods and fields and/or creating and deleting methods and fields are all General Development Tasks. The refactoring ExtractClass thus includes two types of knowledge: Abstract Syntax Tree knowledge as well as General Development Task knowledge. This level of knowledge is incorporated in numerous refactorings, by default. This research will distinguish between refactorings that maintain Abstract Syntax

Tree knowledge and Design Pattern Knowledge only, as all refactorings maintain a certain level of knowledge of general development tasks.

4.2.4c Design Pattern Knowledge

Definition – Design Pattern Knowledge: A change introduced to the structure of an application, and/or the introduction or elimination of a programmable entity, all of which are motivated by a specific design pattern found in a published design pattern catalogue.

More complex refactoring implementations also require knowledge of design patterns. Numerous refactorings provided by [Kerievsky03] (and some provided by [Fowler00, Alur et al.01]) involve complex steps and much more knowledge of the application, class relationships and potential class relationships than is initially apparent. The refactorings are a good guideline, but require a developer's close scrutiny and development skill. Developing tool support for refactorings that map to design patterns requires knowledge of the Abstract Syntax Tree and knowledge of design patterns. Like General Development Tasks are placed in their own packages related only to programmable entities, design pattern knowledge belongs separate from the actual refactoring as it is beyond the requirements of numerous refactorings. Design Pattern Knowledge is different however, from General Development tasks in that the knowledge does not directly concern programmable entities. Rather, the knowledge concerns how to implement a design pattern and attributes required by the design pattern. The refactoring Replace Implicit Tree with Composite [Kerievsky03] is used as an example.

4.2.4c Example

The Composite pattern is a structural one found in [Kerievsky03] and creates a tree structure out of objects, such that the composition of these objects and the objects themselves can be treated similarly. The refactoring Replace Implicit Tree with Composite takes a section of code that would be better represented as a tree, and converts this code to a composite tree structure. It is, however, not that simple. In Figure 4.2.2, taken from [Kerievsky03], one sees that a new class "TagNode" is created and instantiated. This new class is instantiated with a String, its name, and can take in attributes (primitive with a value), can take in objects (possibly itself) and can modify both the attributes and the objects. Thus a tool that performs this refactoring first has to access the Abstract Syntax Tree to determine if the original source is valid (compiles). Then it has to know how to create a new class (General Development Tasks). Finally it has to know how to make the created class a Composite.

```
TagNode orders = new TagNode("orders");
TagNode order = new TagNode("order");
order.addAttribute("number", "123");
orders.add(order);
TagNode item = new TagNode("item");
item.addAttribute("number", "x1786");
item.addValue("carDoor");
order.add(item);
String xml = orders.toString();
```

$\Downarrow$

```
XMLBuilder orders = new XMLBuilder("orders");
orders.addBelow("order");
orders.addAttribute("number", "123");
orders.addBelow("item");
orders.addAttribute("number", "x1786");
orders.addValue("carDoor");
String xml = orders.toString();
```

Figure 4.2.2 Encapsulate Composite with Builder Example Refactoring [Kerievsky03]

This information, the idea of part-whole relationships and uniform treatment of said parts and wholes must be represented somewhere in the tool in order to accomplish the refactoring. This level of knowledge is much more complex and intricate than knowledge of the Abstract Syntax Tree. The knowledge in an Abstract Syntax Tree can display the syntax of a given class or project [Aho et al.98, 49]. Design Pattern Knowledge displays conceptual information about how to implement a design pattern, the class, field and method requirements. Such knowledge is also more thorough than the knowledge of General Development Tasks. General Development Task knowledge displays how to create entities of a programming language. It does not encompass the conceptual purpose of these entities. Design Pattern Knowledge has to encompass purposeful knowledge. That is, it must encompass knowledge that says how to create an entity as well as the purpose of the entity. Thus, a representation of this design pattern knowledge is required in development of a tool that supports refactoring to design patterns. This is the uppermost level of knowledge defined at this time that a refactoring accesses.

4.2.5 Summary of Criteria to Analyze a Refactoring

The following is a table to summarize what has been discussed thus far. A refactoring is now analyzed based upon:

Criterion	***Definition***	
Nested Refactoring	If R is a refactoring and R' is a refactoring, R' is nested inside R if a step in R calls R'. R is said to have a nested refactoring.	
Original Source Scope	The number and the type of one programmable entity surrounding the type of the programmable entity(ies) to be refactored.	
Target Source Scope	The number and type of the largest new programmable entity(ies) created by the refactoring.	
Knowledge Type	The knowledge required to accomplish a refactoring, can be Abstract Syntax Tree knowledge, General Development Task knowledge and/or Design Pattern knowledge.	
	Abstract Syntax Tree Knowledge	A change made to the Abstract Syntax Tree motivated by a programmer's subjective desire to restructure a section of code to enhance readability, comprehension, clarity.
	General Development Task Knowledge	The introduction of a new programmable entity as part of day-to-day programming tasks, often accomplished through the use of tool support in integrated development environments.
	Design Pattern Knowledge	Definition – Design Pattern Knowledge: A change introduced to the structure of an application, and/or the introduction or elimination of a programmable entity, all of which are motivated by one or many aspects of a specific design pattern found in a published design pattern catalogue.

4.3 Justification for Classifications

In examining the available refactorings, various possibilities arose for recognizing similarities among refactorings, but the four that were picked are justified in four ways.

Firstly, a refactoring is a transformation. One can argue indefinitely about the conceptual definition of refactoring as a whole (e.g. cleaning up code, consistent maintenance etc.) but that a single refactoring is a transformation from one state to another, is a difficult point to argue. Thus, it seems only natural to analyze the original state of the system and the target state.

The question immediately becomes then, what measures are valid for analysis? Any sort of usual metric of the original code or target code such as lines of code, function points, depths of nested control statements, etc., is arguably not a valid metric for refactoring. The main issue here is consistency. A refactoring such as Extract Method could be moving one line of code, or it could be moving ten. Yet it functions in the same manner, regardless. The "scope" was the property of choice for the original and target states because it best encompasses all the possible scenarios. At times a refactoring affects lines of code, at times it affects a function point, at times it affects more, but the steps of the refactoring remain the same. As previously stated, the original source code (or target) is a result of, or representation of a concept that needs (needed, respectively) change. Quantifying and qualifying the scope best encompasses this concept. Consequently, the Original Source Scope and Target Source Scope were included in defining similarities among refactorings.

One could then suggest using these metrics on the actual implementation of the refactoring. Not all refactorings are implemented yet however, and as previously stated, the categorization of refactorings (and therefore these principles) is being done in part to facilitate implementation. Thus one avoids a chicken vs. egg situation.

The issues found with traditional metrics in analyzing refactorings gives way to the second justification for the four principles chosen. The original state, measured as the number and type of the scope of the smallest components surrounding the refactoring issue, and the target state, measured as the number and type of the scope of the largest new component created are both *quantifiable* items. They are both specific and finite metrics and relate directly to the refactoring at hand.

The third reason for using these four principles is due to the focus of facilitating implementation for refactorings. The original state tells you what type of component the tool is working with, and how many of these components. The target state tells you how many and what type of component the tool needs to create. This and the knowledge levels give the refactoring developer an idea of what packages the tool will need access to and a general idea of what is involved in implementing the refactoring. As a side effect, by being able to lay out this information, one is better able to estimate development time of the refactoring.

Lastly, conceptually, these principles are clear steps to categorizing refactorings. These principles specify where you are at, where you want to be and most importantly, how you are going to get there.

The benefits of categorizing refactorings have been discussed, as has the basis for the categorization and the reasons for this basis. What follows is a definition and discussion of each of the three categories, examples, and a list of all categorized refactorings. Note that all refactorings likely include access to General Development Tasks. The three categories of refactorings are atomic refactorings, sequential refactorings and finally, complex refactorings.

4.4. Classifications of Refactorings

Please note that for the following definitions, ‘*’ means “many”.

Atomic Refactorings: A refactoring is atomic if the Original Source Scope is no larger than 1 class, the Target Source Scope is smaller than 1 class, the Knowledge Access level is **one** Abstract Syntax Tree and any nested refactorings are themselves atomic.

Original Source Scope <= 1 class $\wedge$
Target Source Scope < 1 class $\wedge$
Knowledge in R = 1 Abstract Syntax Tree $\wedge$
[($(!\exists)$R’ where R’ is nested in R and R’ is a refactoring)
$\vee$
($\forall$R’ where R’ is nested in R, R’ is an atomic refactoring)
]

Sequential Refactoring: A refactoring is sequential if the Original Source Scope is from 1 method, up to and including many classes, the Target Source Scope is from 0 new components up to and including many classes, the refactoring accesses more than one Abstract Syntax Tree, and may include 1 or more atomic and/or sequential refactorings.

```
1 method <= Original Source Scope <= * classes  ∧
0 <= Target Source Scope < * classes  ∧
Knowledge in R =  * Abstract Syntax Trees  ∧
[ (     ∀R’ where R’ is nested in R, R’ is an atomic refactoring ∨
        (!∃)R’ where R’ is nested in R and R’ is a refactoring    )
  ∨
  (     ∀R’ where R’ is nested in R, R’ is an atomic refactoring ∨
        R’ is a sequential refactoring    )
]
```

Complex Refactoring: A refactoring is complex if the Original Source Scope is 1 class or larger, the Target Source Scope is 1 field or larger, the refactoring accesses multiple Abstract Syntax Trees and Design Pattern Knowledge, and finally, the refactoring may include one or more atomic, sequential and/or complex refactorings.

```
Original Source Scope >= 1 classes  ∧
Target Source Scope > 1 field  ∧
Knowledge in R =  * Abstract Syntax Trees and Design Pattern Knowledge  ∧
[ (     ∀R’ where R’ is nested in R, R’ is an atomic refactoring ∨
        R’ is a sequential refactoring ∨
        (!∃)R’ where R’ is nested in R and R’ is a refactoring
  ) ∨
  (     ∀R’ where R’ is nested in R, R’ is an atomic refactoring ∨
        R’ is a sequential refactoring ∨
        R’ is a complex refactoring
  )
]
```

4.5 Explanation and Examples of Refactorings

4.5.1 Atomic Refactorings

Tool support for atomic refactorings is vast and growing rapidly. These are the refactorings that are quick and small: easily clean up your code and perpetuate good programming practices, on a small level. The changes that occur, occur within one file. On Eclipse’s implementation level, one can even see a call to CompliationUnitChange for these types of refactorings, where a CompilationUnit is a single Java file. As a rule of thumb, if the scope of the original source extends as far as one entire class, it is typically due to either a global class variable or a method name that may be changed, not anything more complex. The Target Source has only new variables or new methods. When push comes to shove, some of the atomic refactorings are victim of an overuse of the term refactoring, and could likely be performed manually equally as fast. Verification of the changes can often be made quickly with the naked eye as well. A prime example of this is Rename Method [Fowler00, 273]. Finally, atomic refactorings typically do not include any other refactorings. If they do (e.g. Replace Parameter with Method), the nested refactoring is also atomic. Tool support for these atomic refactorings is still very convenient and extremely useful. The tool support includes error and reference checking. Implemented atomic refactorings can also be used in the implementation of sequential refactorings.

Three examples of atomic refactorings are provided, followed by a list of all atomic refactorings thus far. These examples were taken from Martin Fowler's book on refactoring and can be learned in more detail there.

4.5.1a Extract Method Refactoring (Figure 4.5.1)

The Original Source Scope of Extract Method is 1 method. One is trying to move lines of code into a new method. The lines of code are encompassed inside 1 method as the next largest programmable component. The Target Source Scope is 1 method. Extract Method creates one new method each time it is applied. Lastly, Extract Method accesses only the Abstract Syntax Tree. As explained in section 4.2.1, Extract Method makes three calls to its Abstract Syntax Tree to check that the new method created does not generate any errors. It also does not require any information from the Design Pattern Knowledge. Finally, Extract Method does not include any other refactorings. Due to these principles, Extract Method is an atomic refactoring.

```
void printOwing(double amount){
     printBanner();
    //print details
     System.out.println("name:" + _name);
     System.out.println("amount:" + amount);
}

                              void printOwing(double amount){
                                  printBanner();
                                  printDetails(amount);
                              }

                              void printDetails(double amount){
                                  System.out.println("name:" + _name);
                                   System.out.println("amount:" + amount);
                              }
```

Figure 4.5.1 Extract Method Refactoring Example [Fowler00, 110]

4.5.1b Encapsulate Field Refactoring (Figure 4.5.2)

The Original Source Scope of Encapsulate Field is 1 class. One is trying to give access to a global field via accessor methods. The next largest programmable component surrounding the field is the class. The Target Source Scope is 2 methods, one setter method and one getter method, both under the scope of 1 class as per the definition of atomic refactoring. Lastly, Encapsulate Field makes a call to create two methods, but these methods are specific to the class, encompassed in the class and require only access to the Abstract Syntax Tree. It does not require any information related to design patterns. Finally, it does not require access to any other refactorings. Thus Encapsulate Field is an atomic refactoring.

```
public String _name;

private String _name;
public String getName() { return _name; }
public void setName(String arg) { _name = arg; }
```

Figure 4.5.2 Encapsulate Field Refactoring Example [Fowler00, 206]

4.5.1c Parameterize Method Refactoring (Figure 4.5.3)

The Original Source Scope of Parameterize Method is 1+ methods, within the same class. The refactoring takes similar methods that vary only in a parameter and create a new method that takes in that parameter. The Target Source Scope is therefore 1 method, as per the definition of atomic refactoring. Lastly, the knowledge required to implement this refactoring is again limited to the Abstract Syntax Tree. It does not require knowledge about design patterns nor does it include any other refactorings. Parameterize Method is an atomic refactoring.

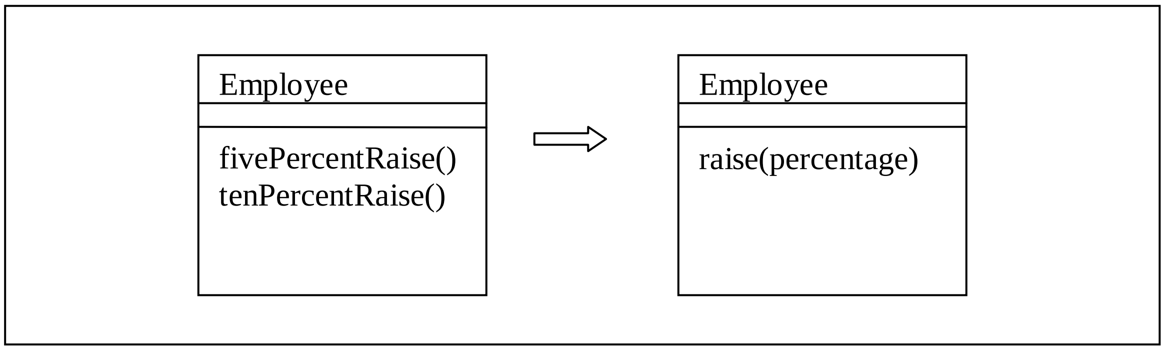


Figure 4.5.3 Parameterize Method Refactoring Example [Fowler00, 283]

The following have also been examined and categorized on similar principles, as atomic refactorings and are taken from [Fowler00].

Add Parameter	Remove Parameter	Separate Query from Modifier
Decompose Conditional	Consolidate Conditional Expressions	Remove Control Flag
Encapsulate Downcast	Replace Error Code with Exception	Replace Exception with Test
Inline Method	Inline Temp	Replace Temp with Query
Introduce Explaining Variable	Split Temp Variable	Remove Assignments to Parameters
Introduce Foreign Method	Hide Method	Self Encapsulate Field
Remove Setting Method	Replace Magic # with Symbolic Constant	Encapsulate Collection
Replace Nested Conditional with Guard Clauses	Introduce Assertion	Rename Method

Replace Parameter with Explicit Methods	Preserve Whole Object	Replace Parameter with Method

Table 4.5.1 Further Atomic Refactorings

4.5.2 Sequential Refactoring

Implementation of sequential refactorings is not as vast as that of atomic refactorings, but can be found to a certain extent in [IntelliJ03]. These refactorings perpetuate good programming practices such as inheritance. The changes that occur, occur within one file, but also develop relationships among classes. These refactorings often incorporate one or more atomic refactorings in their application and are intricate enough that they require execution of test cases to confirm their correctness. These refactorings require some General Development Tasks, namely how to create a class. The target source typically involves the creation /elimination of a new class, although can be less or more. The main difference between atomic and sequential refactorings is that sequential refactorings have access to more than one Abstract Syntax Tree in order to manipulate the relationships between classes. Also, sequential refactorings can be sequences of atomic and/or sequential refactorings that result in requiring knowing how to perform some General Development Tasks and make the refactoring slightly more difficult and time consuming to implement manually. Tool support for these refactorings is extremely convenient and useful. Three examples of sequential refactorings are provided, taken from Martin Fowler’s book [Fowler00]. The diagrams are taken from [Fowler00] and detailed information can be found there.

4.5.2a Extract Class (Figure 4.5.4)

In Extract Class, one creates a new class to handle information that is contained inside one class, but does not necessarily belong in that class. The Original Source Scope of Extract Class is 1 class. There is no one line or section of code that can be clicked on and identified as a problem. There are however, various fields and methods that need to be moved and the next largest surrounding programmable component beyond these fields and methods is the class. The Target Source Scope is also one class: the new class created. While new fields and methods may or may not be created and are likely moved from the original class, a new class is always created and is the largest new programmable entity created by the refactoring. To create the new class the refactoring has to know how to create the class and in which package it goes. It requires no information about design patterns since creating a new class is not directly related to design patterns. Thus the refactoring needs access to the Abstract Syntax Tree (to compile the original source). Finally, after the new class is created, the refactoring applies atomic refactorings Move Method and Move Field (both sequential) refactorings repeatedly until the new class is complete. Due to these four properties, Extract Class is a sequential refactoring.

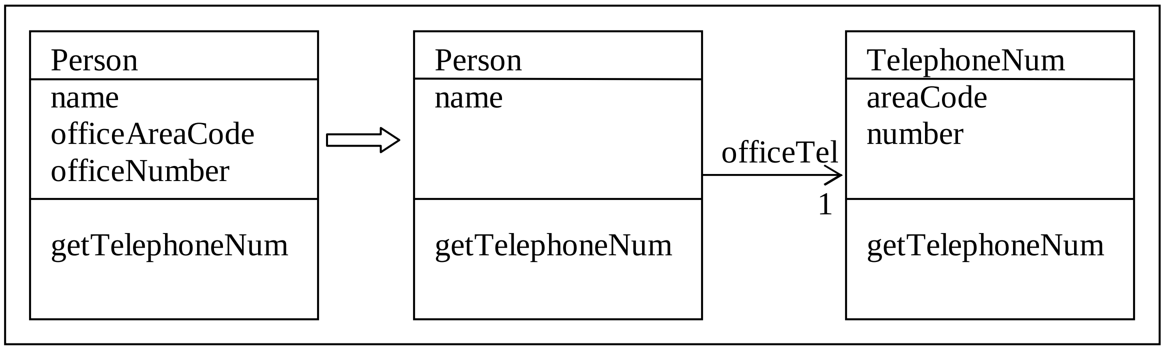


Figure 4.5.4 Extract Class Refactoring Example [Fowler00, 149]

4.5.2b Replace Type Code with Subclass (Figure 4.5.5)

This refactoring creates subclasses for every immutable type in the super class that affects the behaviour of a class. The Original Source Scope of this refactoring is 1 class, because the fields that become classes are encompassed inside only 1 class. The Target Source Scope of this refactoring is one up to many classes. If the original class has *x* immutable fields, then *x* classes are created. To create the *x* new classes, the tool needs to know how to create a class, and it needs to compile the original and target source. Thus, tool support for this refactoring needs access to multiple Abstract Syntax Trees. The refactoring, however, does not incorporate knowledge related to a specific design pattern. It introduces a new class with new attributes, but not for the purpose of corresponding to a given design pattern and therefore does not incorporate design pattern knowledge. Lastly, executing Replace Type Code with Subclass involves the Atomic refactoring Self Encapsulate Field and potentially Push Down Method (sequential) and Push Down Field (sequential) refactorings. Thus, Replace Type Code with Subclass is a sequential refactoring.

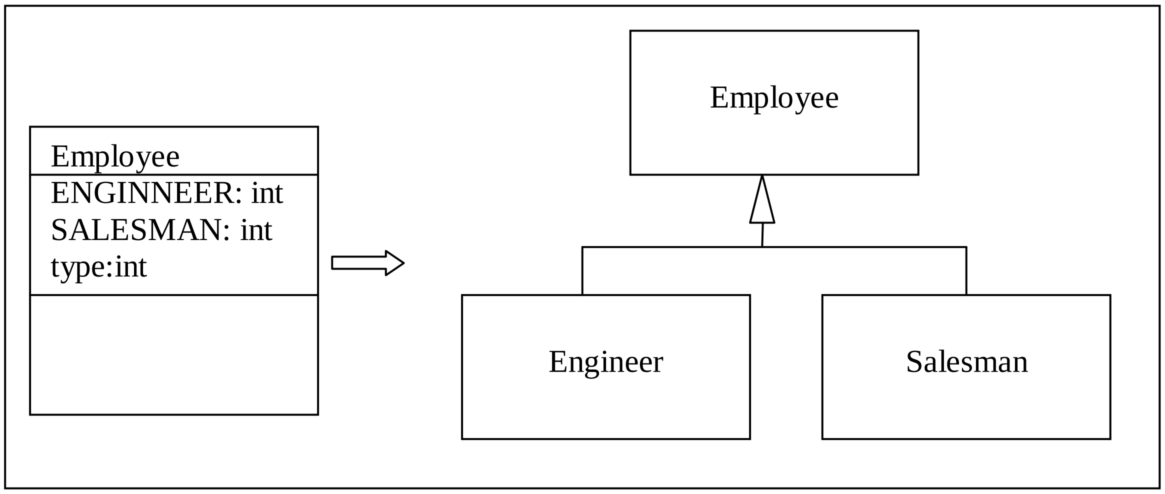


Figure 4.5.5 Replace Type Code with Subclasses Refactoring Example [Fowler00, 223]

4.5.2c Replace Inheritance with Delegation (Figure 4.5.6)

Replace Inheritance with Delegation refactoring is used when a subclass unnecessarily inherits from a superclass some data, where it only needs access to the methods and can delegate these methods to another class, neither super nor sub. The Original Source Scope of this refactoring is two classes. The motivation behind this refactoring lies in the data that both the super and sub classes have access to, specifically, the very fact that both of them have access. Similarly the methods in question are really "part of" both classes due to inheritance. Thus, one has to look at both classes when modifying the fields or methods. The Target Source Scope is 1 field. The relationship between classes is changed. The refactoring does not require any atomic or other sequential refactoring. The refactoring requires access to two Abstract Syntax Tress however, and thus it is classified as a Sequential refactoring.

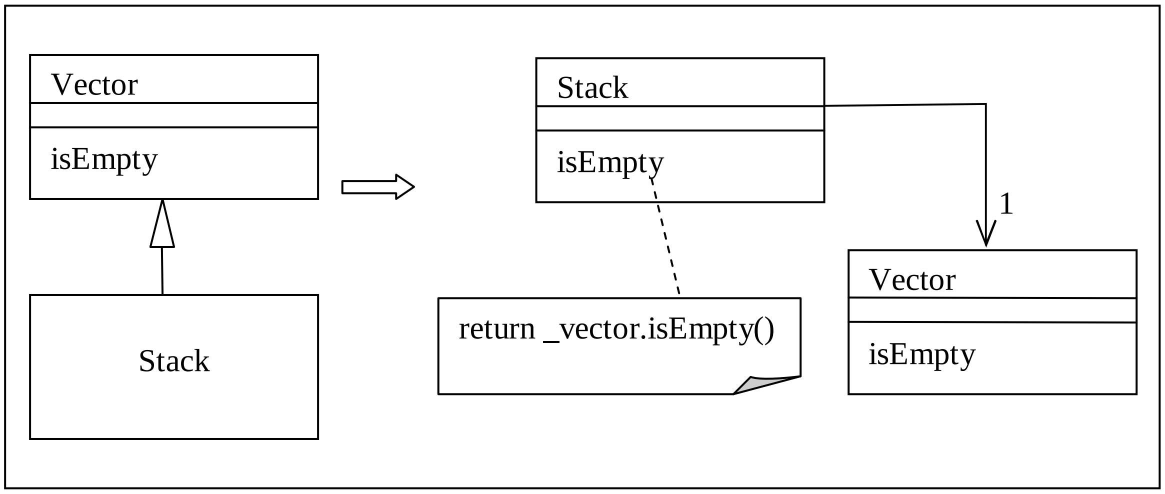


Figure 4.5.6 Replace Inheritance with Delegation Refactoring Example [Fowler00, 352]

The following have also been examined and categorized on similar principles, as sequential refactorings, the bulk of which are taken from [Fowler00].

Replace Method with Method Object	Move Field	Hide Delegate
Remove Middle Man	Replace Data Value with Object	Extract Subclass
Change Unidirectional to Bidirectional	Change Bidirectional to Unidirectional	Replace Record with Data Class
Replace Type Code with Class	Replace Subclass with Fields	Replace Conditional with Polymorphism
Push Down Method	Introduce Parameter Object	Pull Up Field
Pull Up Method	Pull Up Constructor Body	Push Down Field
Extract Superclass	Extract Interface	Collapse Hierarchy
Replace Array with Object	Convert Procedural Design to Objects	Replace Delegation with Inheritance
Move Method	Inline Class	Introduce Local Extension
Replace Enum with Type-Safe Enum [Kerievsky03]		

Table 4.5.2 Further Sequential Refactorings

4.5.3 Complex Refactoring

Complex Refactorings represent the idea mentioned in [GoF95], [Fowler00], that design patterns are targets for refactorings. The changes that occur change relationships between classes, perpetuate the use of design patterns, generate new code and incorporate atomic and sequential refactorings, knowledge from the Abstract Syntax Tree, and General Development Tasks as well as Design Pattern Knowledge. The scope of the original source is typically more than one class and the Target Source Scope is typically at least 1 new class. The key aspects of complex refactorings that differentiate them from atomic and sequential is the amount of new code generated and the design pattern knowledge required to implement the refactoring. Three examples of complex refactoring are provided, followed by a list of complex refactorings from [Kerievsky03], [Alur et al.01], and

[Fowler00]. The refactoring examples are taken from [Kerievsky03] and [Alur et al.01] and more information as to the detail of the refactoring can be found there.

4.5.3a Wrap Entities With Session (Figure 4.5.7)

This refactoring is applied when client files have direct access to entity beans. The Original Source Scope of this refactoring is multiple client files (e.g. Java Server Pages) that have direct access to multiple entity beans. Each client file has to be examined to see if it accesses the entity beans directly. The Target Source Scope is a class: the session that plays the middle man between the client files and the entity beans. New fields and methods may also be created within that session bean, but the session bean is the largest new component and represents the Target Source Scope. Wrapping Entities with Session requires access to Abstract Syntax Trees for the entity beans. It also requires knowing how to create a class (e.g. the session bean) and thus requires access to the General Development Tasks. Finally, it requires knowing how the session bean acts as a middle man between the client files and the entity beans. This is Design Pattern Knowledge: what methods are required in a session bean that wraps entities? How does this change affect calls to the beans originally made in the client files? Question such as these are answered by understanding the design pattern itself and representing that knowledge in the tool. Finally, Wrap Entities in Session requires access to the sequential refactoring Extract Class. Because of these four principles, it is a complex refactoring.

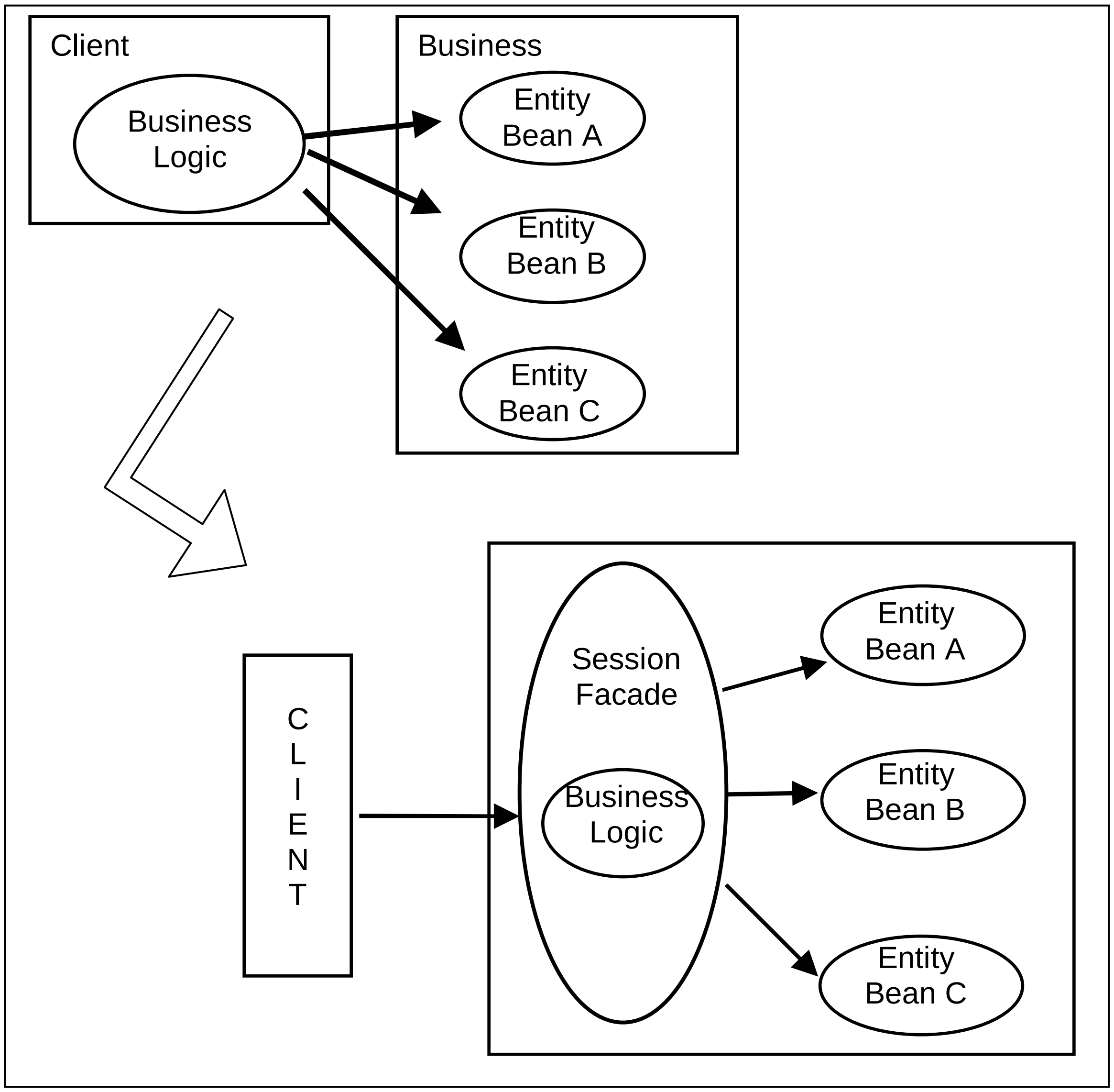


Figure 4.5.7 Wrap Entities with Session Refactoring [Alur et al.01]

4.5.3b Introduce Business Delegate (Figure 4.5.8)

Business Delegate objects are to a Session Façade what a Session Façade is to entity beans: a middle man between client tier and business tier, this time the business tier being a session bean. The Original Source scope of this refactoring is multiple files – multiple client files have access to multiple Session Beans, and each one has to be examined to see if it accesses the Session Bean directly. The Target Source Scope is one class per refactoring application: the Business Delegates that play the middle man between the client and the session. New fields and methods may be created within the Business Delegates, but the class is the largest new programmable entity and thus represents the Target Source Scope. Introducing a Business Delegate requires the Abstract Syntax Tree to compile the project, General Development Tasks to create the classes (the business delegates) and finally Design Pattern knowledge. The Business Delegate pattern found in [Alur et al.01] specifies what the Business Delegate does, its relationship with the other classes in the project, and other design pattern specific information that must be represented in a tool that implements this refactoring. Introduce Business Delegate requires the use of the sequential refactoring Extract Class and because of these four principles, it is a complex refactoring.

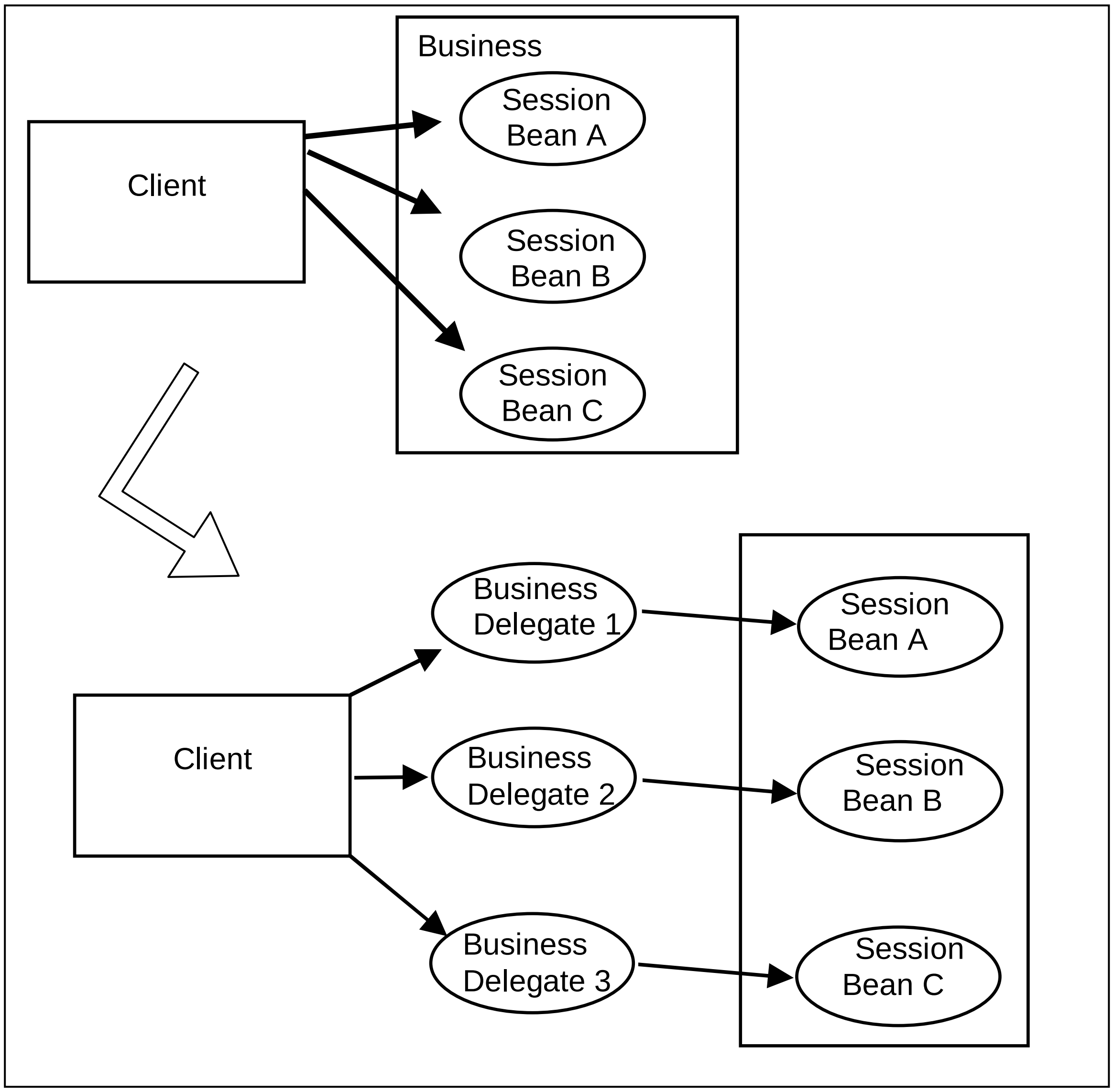
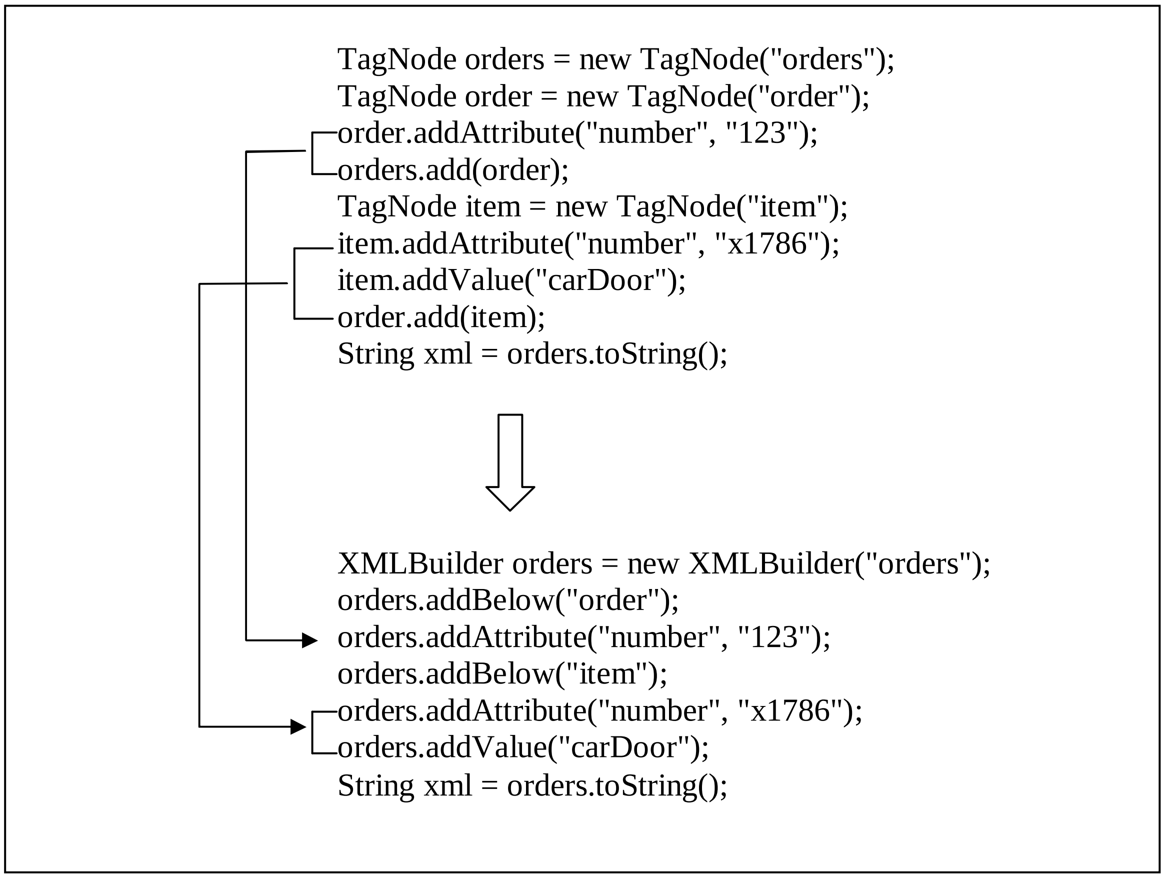


Figure 4.5.8 Introduce Business Delegate Refactoring [Alur et al.01]

4.5.3c Encapsulate Composite with Builder (Figure 4.5.9)

This refactoring is also based on the principle that the presentation logic should be separated from the business logic, specifically when working with a composite design pattern. From [Kerievsky03], code that builds a tree structure often mixes node creation and setup logic with tree creation. A Builder pattern allows the client to focus on building the tree by eliminating the burden of creating the tree. The result is simpler, more intention revealing code and this is the difference between the two pieces of code in Figure 4.5.9. The bottom section of code focuses on building the tree not creating it. Notice that no explicit nodes are instantiated in the second piece of code, instead they are added below the root by saying "**addBelow**". The two lines **orders.add(order)** and **order.add(item)** do not exist in the bottom code. Figure 4.5.9 shows where the bottom code uses one (or two) lines to accomplish what the top code did in two (or three, respectively) lines. The Original Source Scope for this refactoring is one class: the Composite class. The Target Source Scope is also one class: the Builder class. The refactoring requires access to the Abstract Syntax Tree (compiling), access to the General Development Tasks (creating classes such as the Builder) and also access to Design Pattern Knowledge. What relationships are encompassed in the composite pattern? What relationships are encompassed in the Builder pattern? Questions such as these can be found in a representation of the design pattern knowledge. Lastly, this refactoring has access to the Extract Class sequential refactoring. Due to these four principles it is a complex refactoring.



```
TagNode orders = new TagNode("orders");
TagNode order = new TagNode("order");
order.addAttribute("number", "123");
orders.add(order);
TagNode item = new TagNode("item");
item.addAttribute("number", "x1786");
item.addValue("carDoor");
order.add(item);
String xml = orders.toString();
```

```
XMLBuilder orders = new XMLBuilder("orders");
orders.addBelow("order");
orders.addAttribute("number", "123");
orders.addBelow("item");
orders.addAttribute("number", "x1786");
orders.addValue("carDoor");
String xml = orders.toString();
```

Figure 4.5.9 Encapsulate Composite with Builder Example Refactoring [Kerievsky03]

The following are all the complex refactorings found to date from [Fowler00], [Alur et al.01] and [Kerievsky03].

Change Value to Reference [Fowler00]	Introduce Null Object [Fowler00]	Extract Hierarchy [Fowler00]
Replace Constructor with Factory Method [Fowler00]	Extract Composite [Alur et al.01]	Replace Subclass with Visitor [Alur et al.01]
Introduce Polymorphic Creation with Factory Method [Kerievsky03]	Replace Conditional Calculations with Strategy [Kerievsky03]	Replace Implicit Tree with Composite[Kerievsky03]
Encapsulate Composite with Builder [Kerievsky03]	Move Embellishment to Decorator	Move Protection to Proxy [Kerievsky03]
Replace Hard Coded Notifications with Observer [Kerievsky03]	Replace Conditional Searches with Specification [Kerievsky03]	Replace One/Many Distinctions with Composite [Kerievsky03]

Separate Versions with Adapters [Kerievsky03]	Adapt Interface [Kerievsky03]	Replace State-Altering Conditionals with State [Kerievsky03]
Replace Invariable Behaviour with Template Method [Kerievsky03]	Change Reference To Value [Fowler00]	Extract Creation Class [Kerievsky03]
Form Superset Interface [Kerievsky03]	Encapsulate Class with Creation Method [Kerievsky03]	Form Template Method [Fowler00]

Table 4.5.3 Further Complex Refactorings

4.5.4 Summary of Refactoring Classifications

The following table summarizes the definitions of the classifications of refactoring for a refactoring R.

Atomic Refactoring
Original Source Scope <= 1 class ^ Target Source Scope < 1 class ^ Knowledge in R = 1 Abstract Syntax Tree ^ [((!∃)R' where R' is nested in R and R' is a refactoring) ∨ (∀R' where R' is nested in R, R' is an atomic refactoring)]
Sequential Refactoring
1 method <= Original Source Scope <= * classes ^ 0 <= Target Source Scope < * classes ^ Knowledge in R = * Abstract Syntax Trees

```
$\wedge$
[ ( $\forall$R’ where R’ is nested in R, R’ is an atomic refactoring $\vee$
    (!$\exists$)R’ where R’ is nested in R and R’ is a refactoring )
  $\vee$
  ( $\forall$R’ where R’ is nested in R, R’ is an atomic refactoring $\vee$
      R’ is a sequential refactoring )
]
nb: ‘*’ means many
```

Complex Refactoring

```
Original Source Scope >= 1 classes
$\wedge$
Target Source Scope > 1 field
$\wedge$
Knowledge in R = * Abstract Syntax Trees and Design Pattern Knowledge
$\wedge$
[ ( $\forall$R’ where R’ is nested in R, R’ is an atomic refactoring $\vee$
      R’ is a sequential refactoring $\vee$
      (!$\exists$)R’ where R’ is nested in R and R’ is a refactoring )
  $\vee$
  ( $\forall$R’ where R’ is nested in R, R’ is an atomic refactoring $\vee$
      R’ is a sequential refactoring $\vee$
      R’ is a complex refactoring )
]
```

4.6 Chapter Summary

This chapter has covered numerous definitions pertinent to refactoring. The benefits of providing refactoring categorizations were discussed as being organization, understanding, task estimation and implementation. The criteria for classifying refactorings were given as the Original Source Scope, the Target Source Scope, Knowledge Type and Nested Refactorings. Definitions for each were provided, and Knowledge Type was recognized as the primary criterion for classifying refactorings. There are three types of knowledge a refactoring can have: knowledge of the Abstract Syntax Tree, knowledge of General Development Tasks and knowledge of Design Patterns. Knowledge of design patterns is the key element that distinguishes complex refactorings from all other refactorings. Three classifications of refactoring were given. Atomic refactorings do not extend past a single Abstract Syntax Tree, Sequential refactorings deal with multiple Abstract Syntax Trees and Complex refactorings deal with multiple Abstract Syntax Trees and design pattern knowledge. An explanation of each classification was given with three examples per classification.

5 TOOL IMPLEMENTATION

The Design Pattern Developer tool was created in Eclipse, an open source integrated development environment produced by IBM [Eclipse03]. A previously stated, it was chosen as the development environment for this research due to its availability of code and plug-in development capabilities. The Design Pattern Developer tool is a seven page wizard that takes an initial Java 2 Enterprise Edition project and changes servlet files, java server pages and entity beans to match a J2EE design pattern. The following chapter is divided into five sections. Section one takes the reader through a demo of the tool and section two looks at the change introduced by the tool. The Tool Architecture section discusses the design of the tool with a conceptual look at the packages in the tool. Section four looks at implemented complex refactorings in the tool. The chapter concludes with a summary. Implementation details can be found in Appendix B.

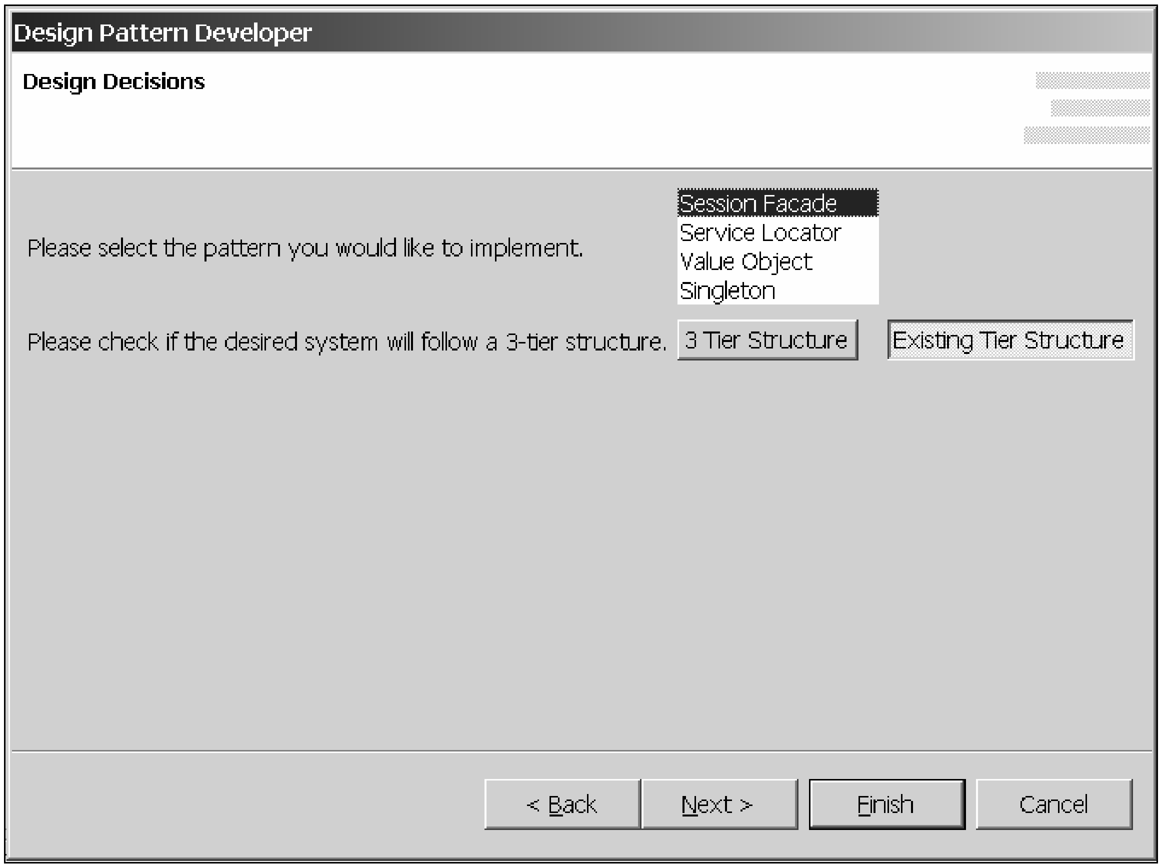
5.1 Session Façade with Value Object Demonstration

What follows is a demonstration of the tool when applying the Session Façade with the Value Object design pattern, one of the four patterns provided by the tool. The wizard is divided into two sections, the first three pages and the last four. The first three pages are design decisions the user of the tool must make before applying the refactorings. The last four pages are the actual application of the refactorings to the J2EE application.

5.1.1 Tool Design Decisions

<u>5.1.1.1 Design Decisions A (Figure 5.1.1)</u>

Page A is the first of the 3 design decision pages and asks one key question of the user: what design pattern will be implemented? Here Session Façade is selected and a second question appears. This asks if the new application will follow the existing hierarchy structure, or a new three tier structure. This new 3-tier structure would be used in a worst case scenario where the initial application is not well structured. Unrelated classes may not be grouped into separate packages, for example. For example purposes, the user here leaves “Existing Tier Structure” selected (as default) and presses Next. If the “New 3 Tier Structure” button had been pressed three new packages would be created and the selected files (as chosen in the page presented in Figure 5.1.4) would be placed into their respective packages.


Design Pattern Developer

Design Decisions

Please select the pattern you would like to implement.

Session Facade
Service Locator
Value Object
Singleton

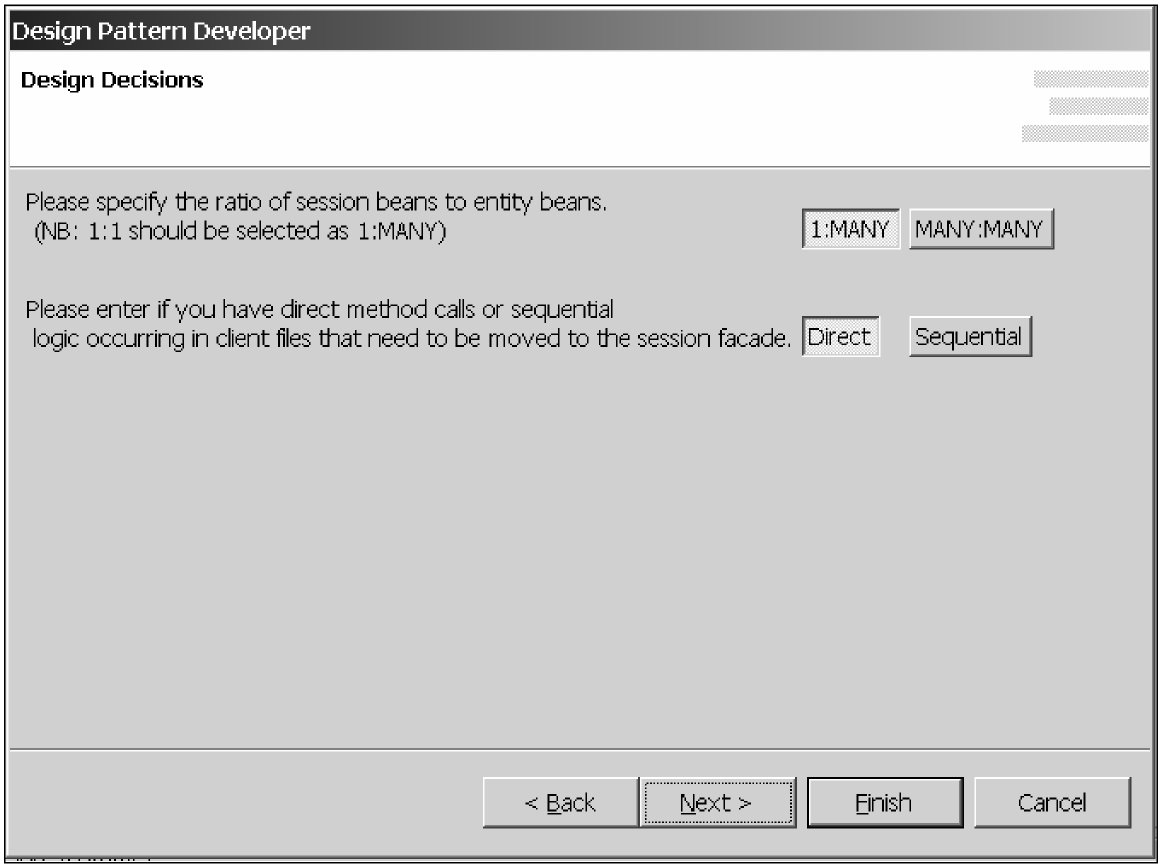
Please check if the desired system will follow a 3-tier structure.

Figure 5.1.1 Tool Design Decisions Page A

<u>5.1.1.2 Design Decisions B (Figure 5.1.2)</u>

Page B poses two questions to the user. First is to specify the ratio of session to entity beans. The developer has one of two options: 1:MANY or MANY:MANY. When implementing the session façade pattern one may want multiple session beans for groups of entity beans. This would best be applied in a large J2EE application where the developer wishes to organize the workflow in numerous Session Facades. In such a case the developer would select the MANY:MANY option. The default is set to 1:MANY, which creates one session bean for all entity beans. Note the scenario of 1:1 is covered by the scenario of 1:MANY and the scenario of MANY:1 is not a realistic one as having only one entity bean typically does not occur. Here again, the default of 1:MANY is left selected.

The second question asks if the client files have direct or sequential logic that needs to be moved to the Session Façade. Direct Logic refers to only one line of code in the client (or integration) files that make a direct reference to an entity bean. Sequential logic refers to multiple lines of code. A prime example would be calling a finder method on an entity bean and then filtering through all entity beans to manually pull out specific beans. Such filtering logic needs to be moved to the session bean. This is not as easy as moving a single line of code as multiple java server page lines of code may contain html tags inside the java code. Here again, the default of direct logic is left selected and the user clicks Next.


Design Pattern Developer

Design Decisions

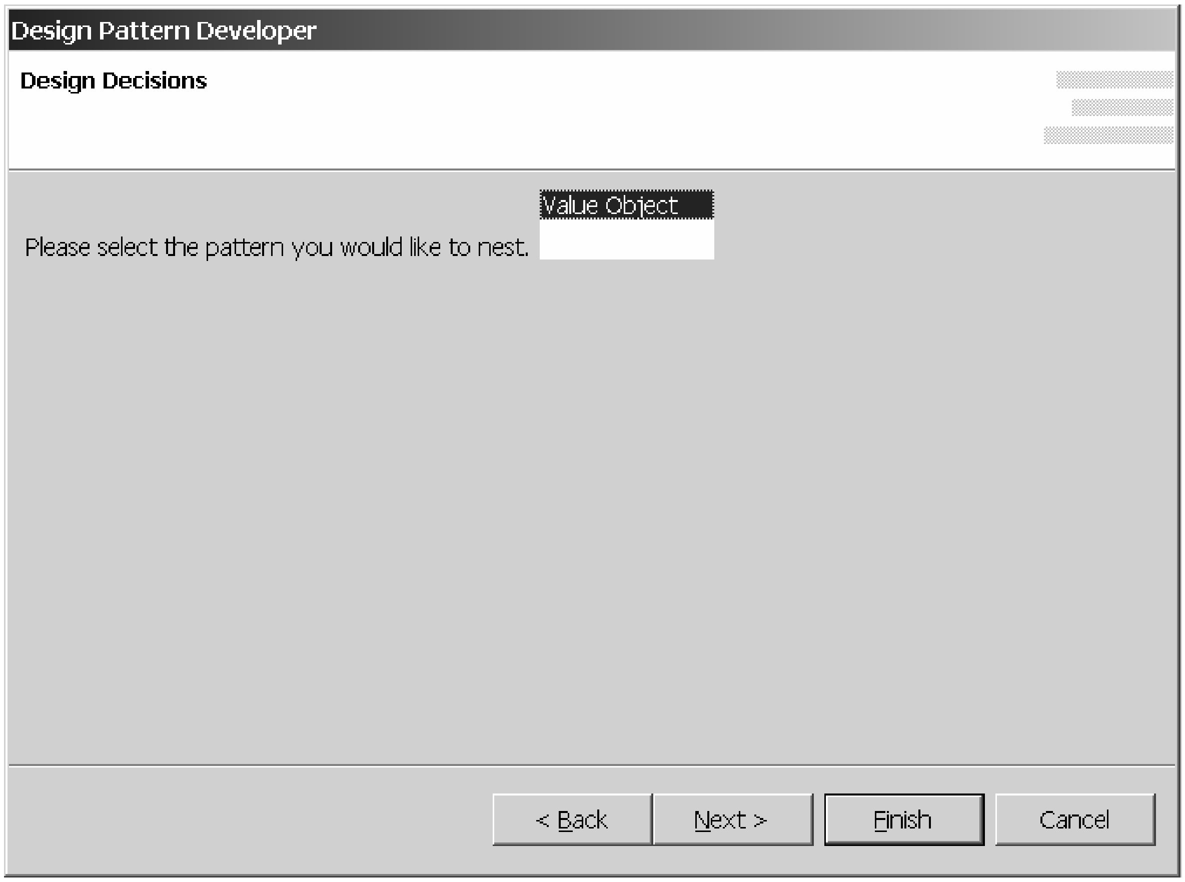
Please specify the ratio of session beans to entity beans.
(NB: 1:1 should be selected as 1:MANY)

Please enter if you have direct method calls or sequential logic occurring in client files that need to be moved to the session facade.

Figure 5.1.2 Tool Design Decisions Page B

<u>5.1.1.3 Design Decisions C (Figure 5.1.3)</u>

The last of the design decision pages only appears when selecting a session façade design pattern to implement. Page C asks which patterns should be nested inside the Session Façade and gives Value Object as options. The user selects the Value Object pattern and clicks Next to go to the implementation stage of the wizard.



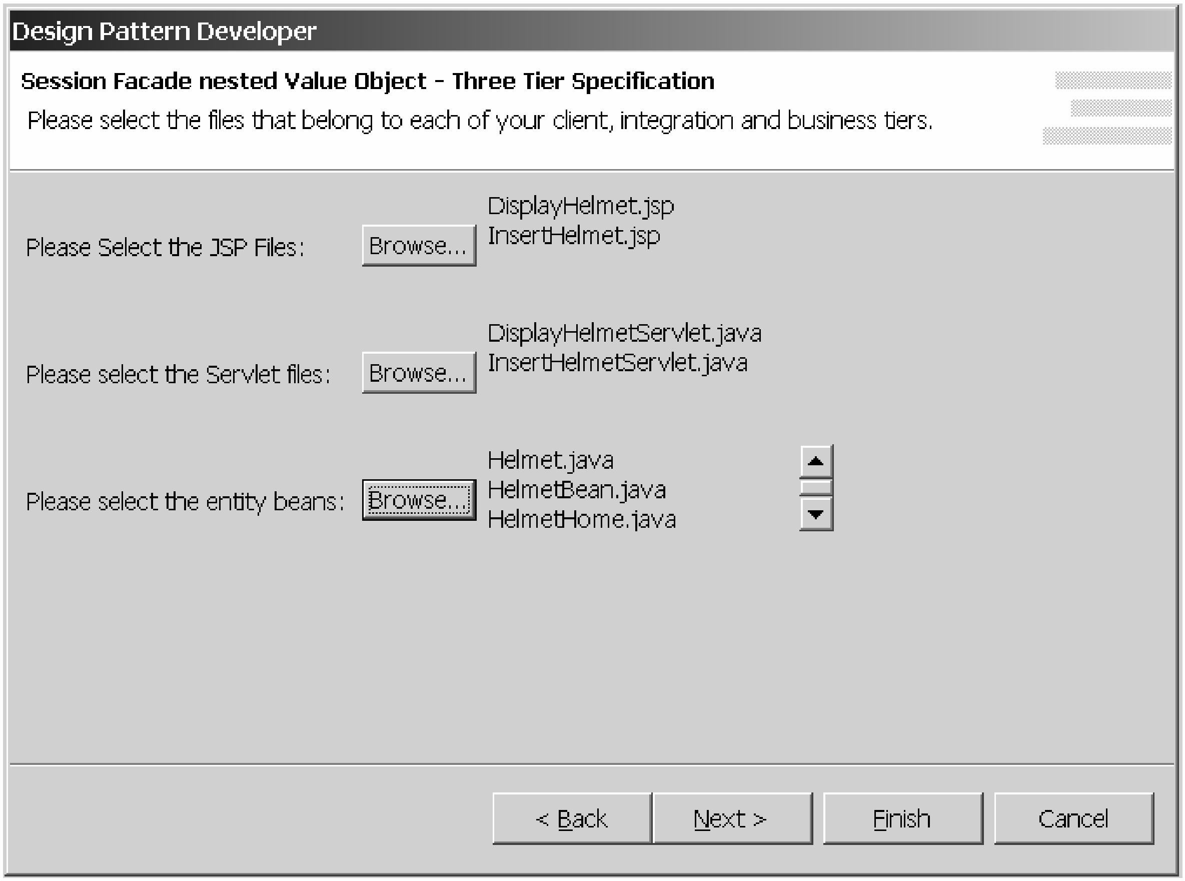
Please select the pattern you would like to nest. Value Object

Figure 5.1.3 Tool Design Decisions Page C

5.1.2 Tool Refactoring

5.1.2.1 Page 1 (Figure 5.1.4)

Page 1 of the implementation part of the Design Pattern Developer asks the user to identify the java server pages, servlet files and entity beans to be analyzed. Only the selected files will be refactored. The user selects the Browse buttons and selects the desired files as shown in Figure 5.1.4 and clicks Next.


Design Pattern Developer

Session Facade nested Value Object - Three Tier Specification

Please select the files that belong to each of your client, integration and business tiers.

Please Select the JSP Files: Browse... DisplayHelmet.jsp InsertHelmet.jsp

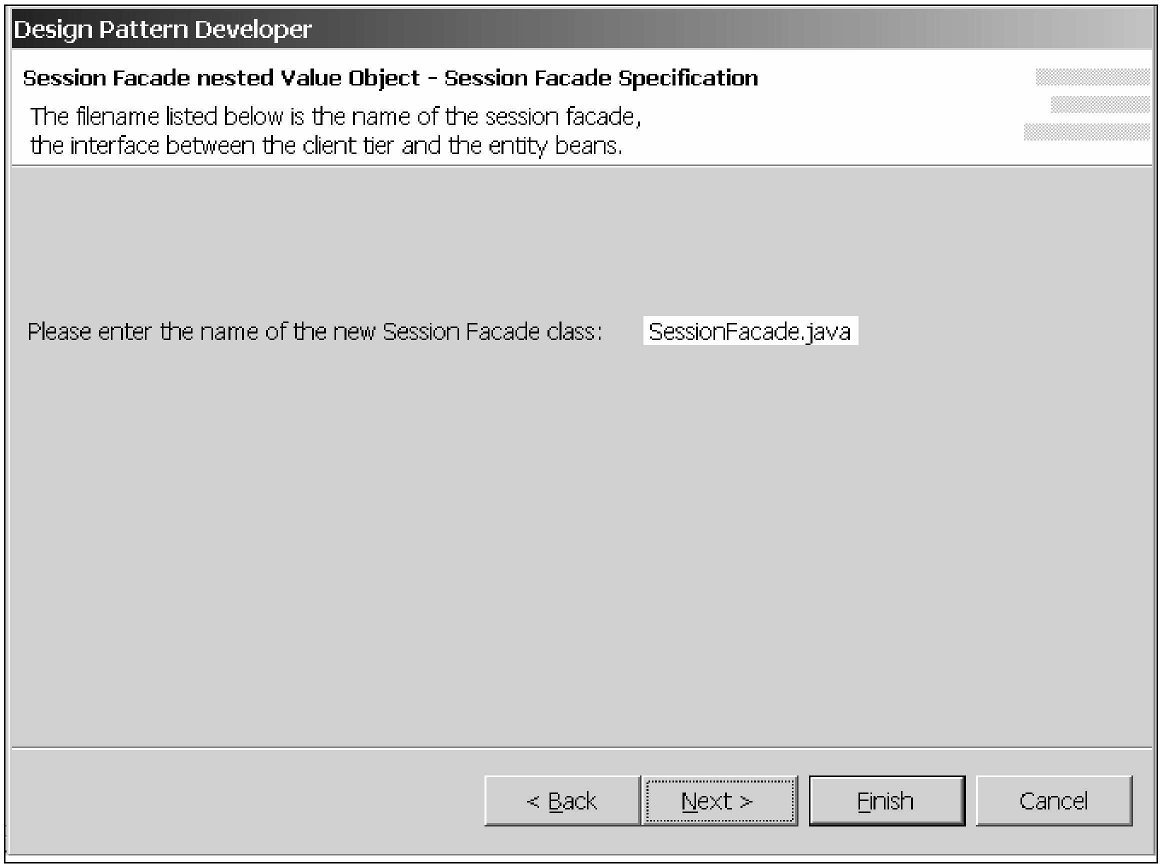
Please select the Servlet files: Browse... DisplayHelmetServlet.java InsertHelmetServlet.java

Please select the entity beans: Browse... Helmet.java HelmetBean.java HelmetHome.java

Figure 5.1.4 Tool Refactoring Page 1

5.1.2.1 Page 2 (Figure 5.1.5)

At page 2 the first complex refactoring is called. The user enters the name of the session bean to be created (the bean that will be the Session Façade) and the tool creates the corresponding files. An alternative option is shown if the user had selected the MANY:MANY button at Page B of the design decisions. In this case Page 2 takes in the number of session beans to create. It also takes in the names of the requisite number of session beans and generates the requisite number of files. Finally, it asks the user to select the entity beans that correspond to each session bean. In the 1:MANY situation shown in Figure 5.1.5, the user enters the name and clicks Next.



Design Pattern Developer

Session Facade nested Value Object - Session Facade Specification

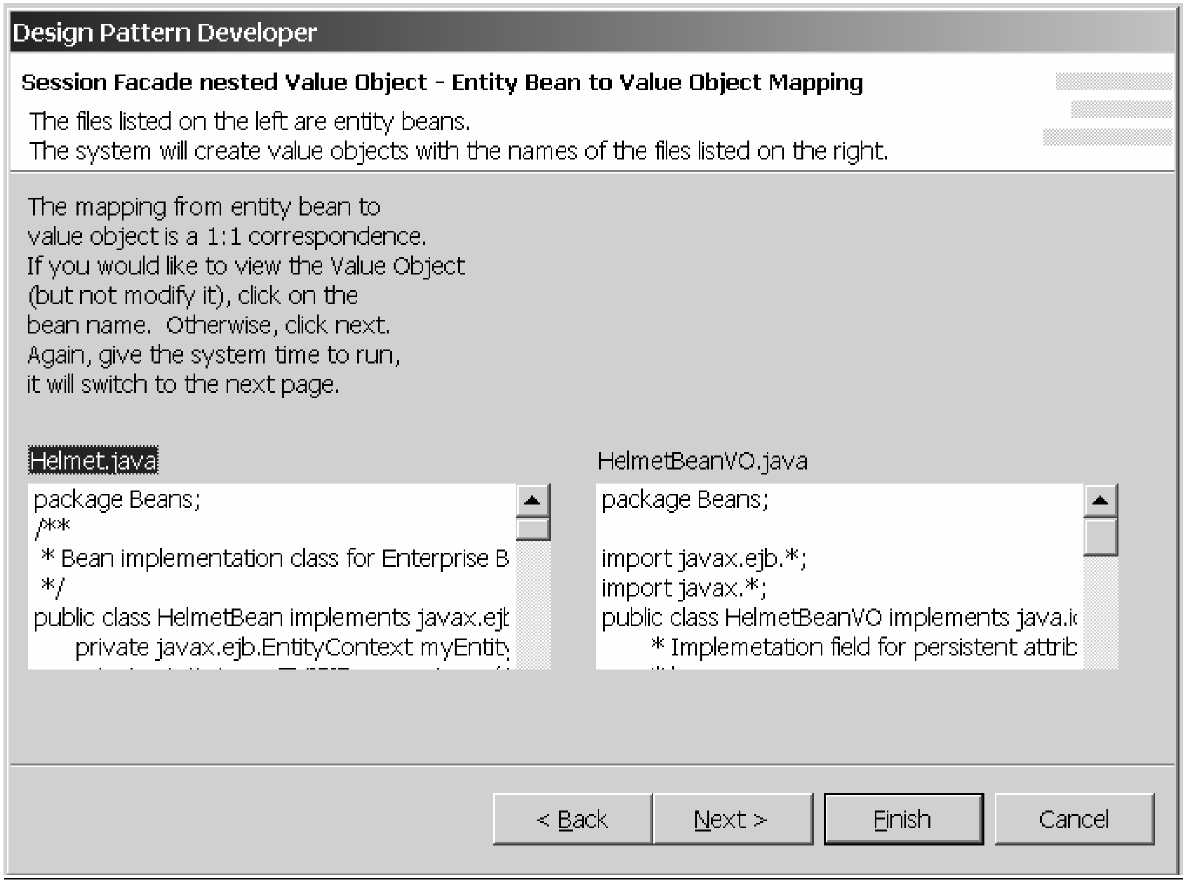
The filename listed below is the name of the session facade, the interface between the client tier and the entity beans.

Please enter the name of the new Session Facade class: SessionFacade.java

Figure 5.1.5 Tool Refactoring Page 2

5.1.2.3 Page 3 (Figure 5.1.6)

When arriving onto page three the second complex refactoring is called. The value objects corresponding to the selected entity beans are all created and the modification to the entity bean is performed. Page 3 is a display page to show the changes and the new value object created. By scrolling down the left pane one sees the entity bean code with the added method “getValueObject”. On the right pane one sees the code for the value object. The user clicks Next to go to the last page of the wizard.


Design Pattern Developer

Session Facade nested Value Object - Entity Bean to Value Object Mapping

The files listed on the left are entity beans.
The system will create value objects with the names of the files listed on the right.

The mapping from entity bean to
value object is a 1:1 correspondence.
If you would like to view the Value Object
(but not modify it), click on the
bean name. Otherwise, click next.
Again, give the system time to run,
it will switch to the next page.

Helmet.java

```
package Beans;
/**
 * Bean implementation class for Enterprise B
 */
public class HelmetBean implements javax.ejt
     private javax.ejb.EntityContext myEntity
```

HelmetBeanVO.java

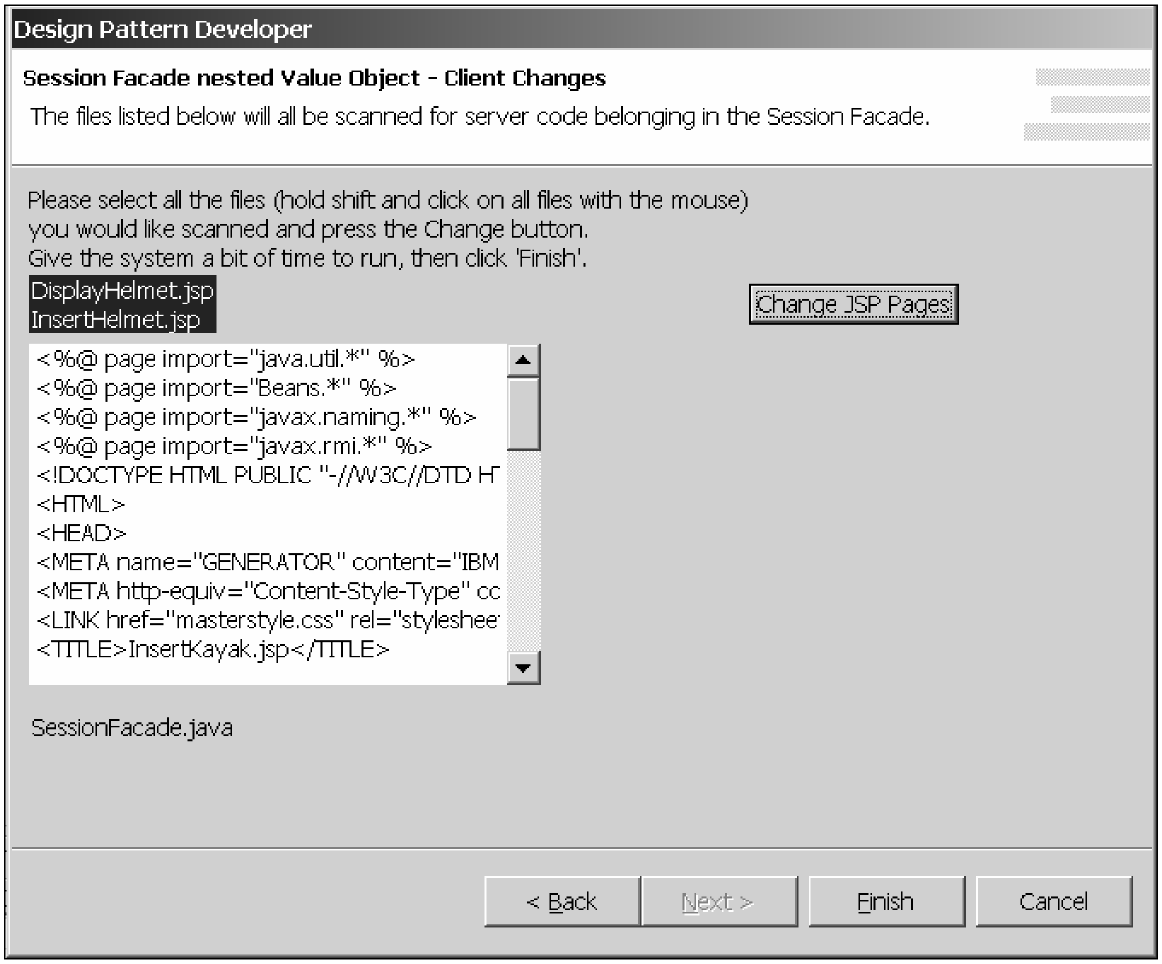
```
package Beans;

import javax.ejb.*;
import javax.*;
public class HelmetBeanVO implements java.i(
      * Implemetation field for persistent attrib
```

Figure 5.1.6 Tool Refactoring Page 3

5.1.2.4 Page 4 (Figure 5.1.7)

Here the java server pages are modified. The user selects the files to be changed and clicks the “Change JSP” button. The servlets are modified upon arrival onto page 4. If the user had selected the MANY:MANY option on page B of the design decisions, page 4 would be altered. The user would be required to select which Session Façade corresponds to the java server page. Also, if the user selects Sequential Logic on page B of the design decisions pages, here at page 4 the user highlights and modifies the java server page lines of code to be changed. With the default selections the user can click the “Change JSP Pages” button, then click Finish.



Session Facade nested Value Object - Client Changes

The files listed below will all be scanned for server code belonging in the Session Facade.

Please select all the files (hold shift and click on all files with the mouse)
you would like scanned and press the Change button.
Give the system a bit of time to run, then click 'Finish'.

DisplayHelmet.jsp
InsertHelmet.jsp

```
<%@ page import="java.util.*" %>
<%@ page import="Beans.*" %>
<%@ page import="javax.naming.*" %>
<%@ page import="javax.rmi.*" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD H
<HTML>
<HEAD>
<META name="GENERATOR" content="IBM
<META http-equiv="Content-Style-Type" cc
<LINK href="masterstyle.css" rel="styleshee
<TITLE>InsertKayak.jsp</TITLE>
```

SessionFacade.java

Figure 5.1.7 Tool Refactoring Page 4

5.1.3 Further Tool Workflow

The Design Pattern Developer has four design patterns implemented and one more combination design pattern. The initial four that appear on the first page of the wizard are Session Façade, Value Object, Singleton and Service Locator. If Session Façade is selected as the design pattern of choice, the user has the option to incorporate a Value Object pattern. As shown in the example above, the tool then creates a session bean that is the Session Façade between the client (presentation) files and the entity beans. The tool generates workflow methods in the Session Façade that correspond to calls to the entity beans that the client files make. Next, the Design Pattern Developer changes the calls in the presentation files that were made directly to the entity bean, to be calls to the newly created Session Façade methods. If the Value Object pattern was selected to incorporate in the Session Façade pattern, the tool creates one Value Object for every entity bean selected by the user. The data in the Value Object are the fields from the entity bean and accessor methods for each field. Lastly, the tool creates a method in the entity bean to return the Value Object when the bean is called. The Service Locator pattern is incorporated automatically in the Session Façade pattern because the purpose of the Session Façade encompasses the purposes of the Service Locator pattern. That is to say, the service locator information (in the chosen domain, the Java Naming and Directory Lookup information [J2EE, 176]) is included in the Session Façade. Again, the diagram in Figure 5.2.1 and 5.2.2 show the change created by the Session Façade Value Object pattern.

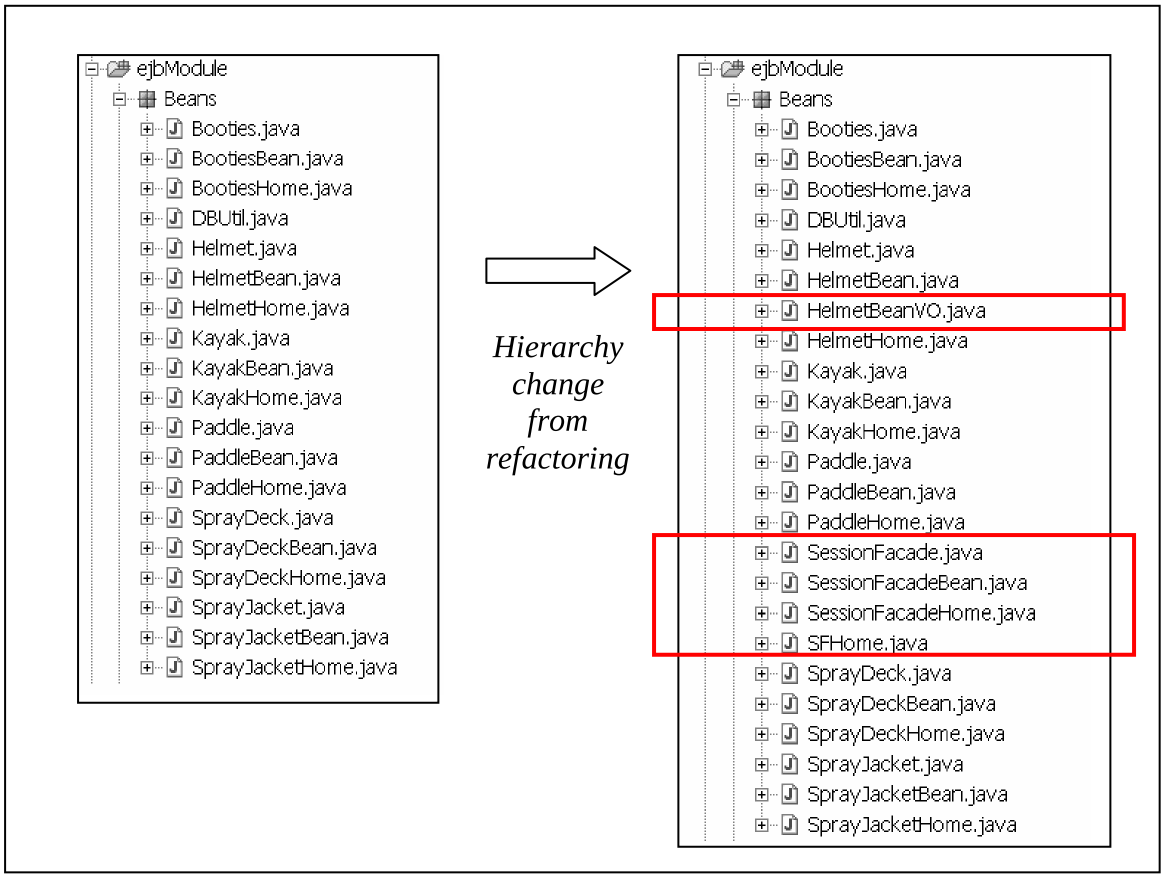
If the user chooses to implement the Value Object pattern, one value object is created for every entity bean selected. The data in the Value Object are the fields from the entity bean and accessor methods for each field. The tool also creates a method in the entity bean to return the Value Object when the bean is called.

If the user chooses the Singleton design pattern, the user has the option to create a new Singleton class or to convert an existing class to a Singleton class. In the former situation, the user enters the name of the class to be created and the class is created with a private constructor and a "**getInstance**" method, which returns the first and only instance of the class created during runtime. In the latter situation, the class selected is modified. The constructor is made private and a "**getInstance**" method is added to return the first and only instance of the class created during runtime.

All changes to java files are made using the built in Abstract Syntax Tree in eclipse. Creating a method or modifying a method creates a node or modifies a node in the Abstract Syntax Tree, respectively. Changes made to the Java Server Pages are done through text parsing.

5.2 The Result of the Change

The result of the application from the demonstration in Section 5.1, is shown in Figure 5.2.1 and Figure 5.2.2. A new value object was created, corresponding to the entity bean selected. A new session bean was created as the session façade. Finally, the java server pages, entity beans and servlets were modified.



```
ejbModule
  Beans
    Booties.java
    BootiesBean.java
    BootiesHome.java
    DBUtil.java
    Helmet.java
    HelmetBean.java
    HelmetHome.java
    Kayak.java
    KayakBean.java
    KayakHome.java
    Paddle.java
    PaddleBean.java
    PaddleHome.java
    SprayDeck.java
    SprayDeckBean.java
    SprayDeckHome.java
    SprayJacket.java
    SprayJacketBean.java
    SprayJacketHome.java
```

```
ejbModule
  Beans
    Booties.java
    BootiesBean.java
    BootiesHome.java
    DBUtil.java
    Helmet.java
    HelmetBean.java
    HelmetBeanVO.java
    HelmetHome.java
    Kayak.java
    KayakBean.java
    KayakHome.java
    Paddle.java
    PaddleBean.java
    PaddleHome.java
    SessionFacade.java
    SessionFacadeBean.java
    SessionFacadeHome.java
    SFHome.java
    SprayDeck.java
    SprayDeckBean.java
    SprayDeckHome.java
    SprayJacket.java
    SprayJacketBean.java
    SprayJacketHome.java
```

Figure 5.2.1 Result of Refactoring

Figure 5.2.1 is a screen shot of the actual classes in eclipse. The left set of classes is the pre-refactoring code, the right side is the post-refactoring code. Note the addition of 5 new classes created by the tool to support the Session Façade with a Value Object design pattern. HelmetBeanVO.java is the value object corresponding to the Helmet entity bean, the three SessionFacade classes make up the session bean that is the Session Façade. SFHome.java is a utility class created to access the session bean.

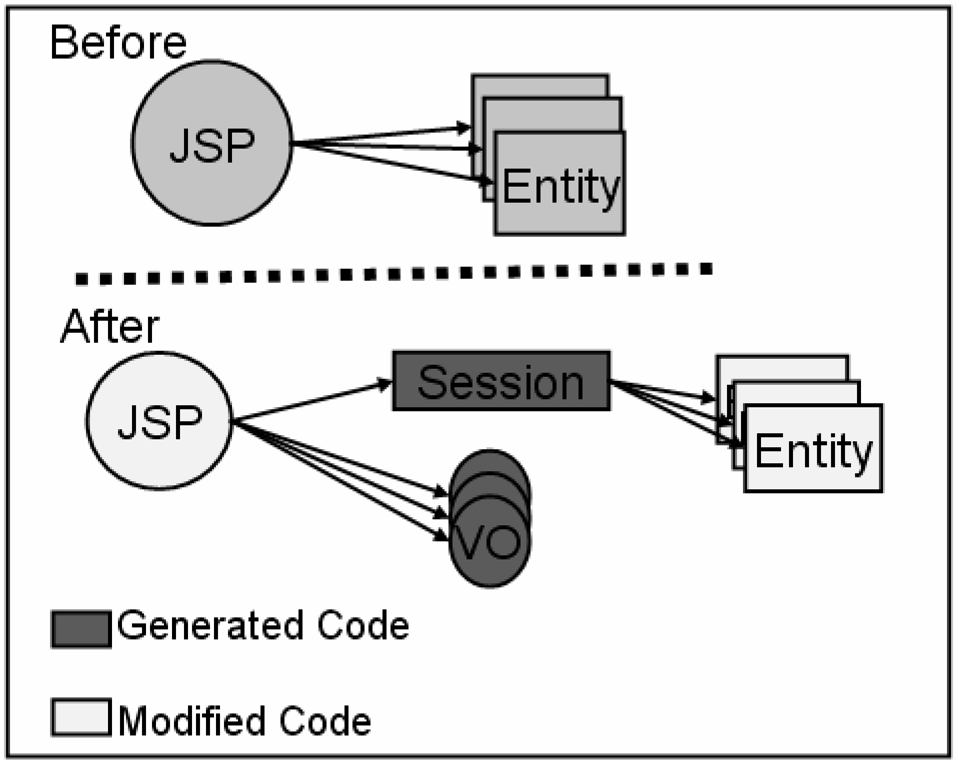


Figure 5.2.3 Change Introduced by the Design Pattern Developer

Figure 5.2.3 is a pictorial representation of the change that occurred by applying the Session Façade with a Value Object design pattern. Note the legend identifies the components that are created by the tool in dark – the value object(s) and session façade. The modified code (entity beans and java server pages) are identified by in light.

5.3 Tool Architecture Overview.

Figure 5.3.1 shows the tool architecture of the DesignPattern Developer. The wizard package accesses the Original Project Analysis package and the Target Project Analysis package. The Original Project Analysis package accesses the Rule Store. The Target Project Analysis package accesses the Rule Store. The Target Project Analysis package accesses the Original Project Analysis package and the Rule Store. At the implementation level there are more actual java packages than these four, but each package corresponds to one of these conceptual packages. Implementation details can be found in Appendix B

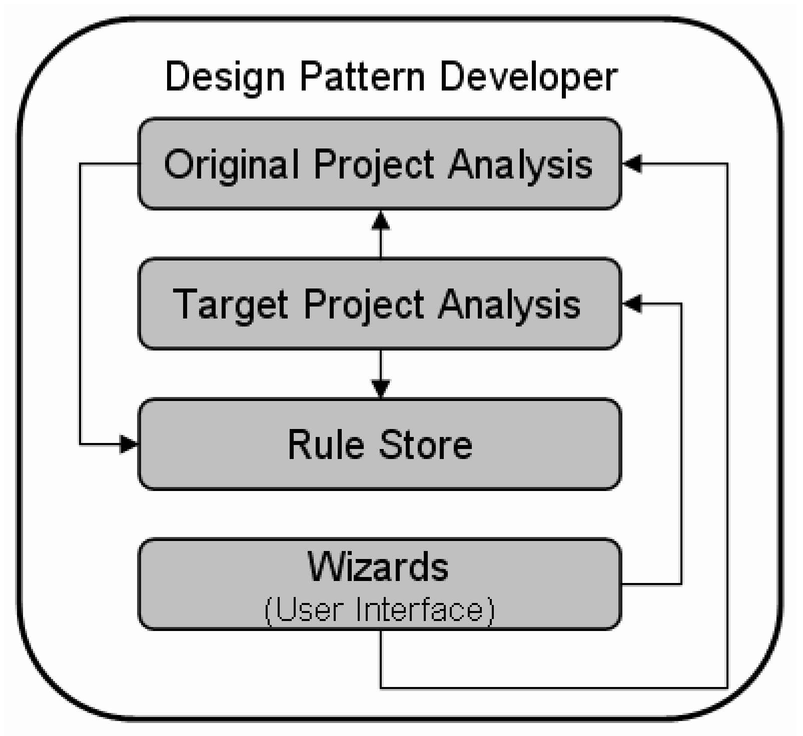


Figure 5.3.1 Tool Architecture

5.3.1 Package Description

Each wizard is implemented using four packages, each of which is described below. Three packages represent three of the four attributes by which a refactoring is analyzed: Original Source Scope, Target Source Scope, and Knowledge Access Level. The last package is the wizard functionality. Each package is described in general first. A class description for each package in each wizard is given in Appendix B. Figure 5.3.1 shows a diagram of the tool's architecture. One should note that the following three packages are conceptual packages. In terms of implementation, they were broken down into more packages necessary for each specific pattern.

5.3.1.1 Original Project Analysis

The Original Project Analysis package is representative of the Original Source Scope. It is the reason behind applying the refactoring. The classes in the Original Project Analysis package of each wizard look for the issues in the original project and prepare the original project for modification. This package must have access to the knowledge of the design patterns, in order to find issues in the files. It also incorporates eclipse packages that assist in file manipulation and general development tasks.

5.3.1.2 Target Project Analysis

The Target Project Analysis package is representative of the Target Source Scope in that it is what actually produces the final product. The Target Project Analysis package contains all the Complex Refactorings and has access to the Original Project Analysis package and knowledge of the design patterns. Also, the Target Project Analysis package has access to eclipse's file manipulation and java file modification packages.

5.3.1.3 Rule Store

The Rule Store represents the knowledge to which a refactoring has access. There is only one shared Rule Store for all wizards implemented and it maintains knowledge of the J2EE domain, design patterns and general organizational knowledge.

5.3.1.4 Wizard

The final package category is the wizard package which instantiates three initial pages then the design pattern specific pages. The first three pages are design decision pages and ask the user to answer a few questions before implementing the pattern. These will be discussed further in section 5.5.1.

5.4 Implemented Complex Refactorings

The Target Project Analysis package for each wizard contains all complex refactorings required to implement the given design pattern. While each Target Project Analysis contains some helper classes, the focus of this writing lies in the specific complex refactoring and what follows is a description of the complex refactorings implemented in the Design Pattern Developer tool

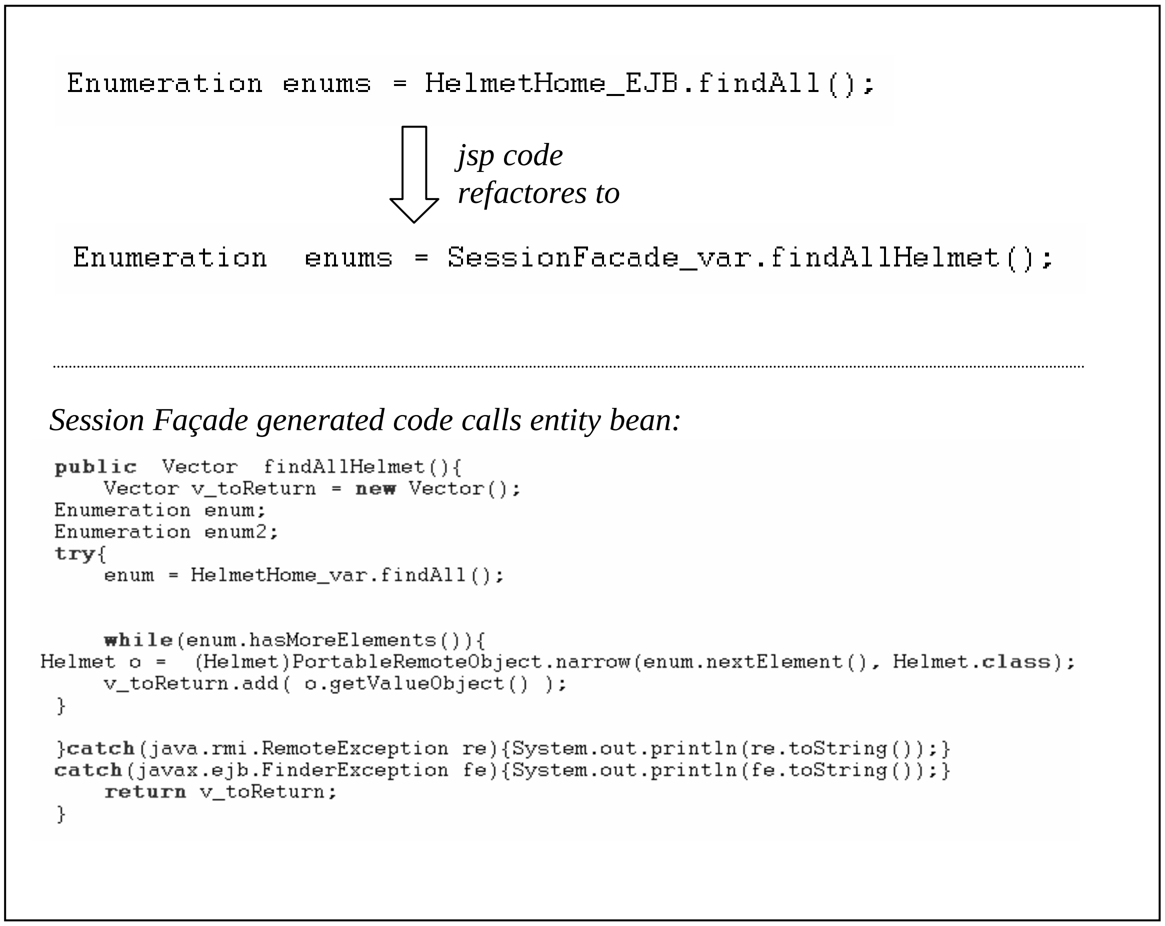
5.4.1 Session Façade Value Object Refactorings

In the Session Façade with a Value Object pattern the key complex refactorings are CreateSessionFacadeCRefactoring and CreateValueObjectCRefactoring.

CreateSessionFacadeCRefactoring is an implementation of the complex refactoring Wrap Entity with Session, from section 4.5.3a. The Original Source Scope is multiple files and multiple file types. This is shown by the corresponding Original Project Analysis package that accesses servlets, java server pages, the entire project structure and information from the user. The Target Source Scope is again multiple files – as will be shown by the change this complex refactoring introduces. The Design Pattern Knowledge is shown in the import lines that import four files from the Rule Store: ProjectStructureRules.java, JavaClassRules.java, SFUtilityClass.java and SessionBeanTemplate.java. The task of this refactoring is to introduce a new session bean into the project and modify the client and integration files (java server pages and servlets respectively).

The refactoring is created with the name of the new session bean to create. The refactoring creates an instance of SessionBeanTemplate, which creates the actual remote interface, home interface and bean implementation files. A utility class is then created using SFUtility.java, so that the same instance of the session is always accessed. This was a design choice made during development of the tool and is not necessarily required. New package declarations or imports are placed in the new session bean files created.

The CreateSessionFacadeCRefactoring requires the assistance of another complex refactoring, CreateSFBeanHomeReferencesCRefactoring.java. This file gets the files corresponding to the home and remote interfaces, and the bean implementation and establishes references to the chosen entity beans inside the session bean implementation. The java server pages and servlet files are changed to reference this new session bean. In the session bean, any finder or create methods found in the java server pages or servlet files are moved to the session bean and wrapped in a new method that is now called from the java server page. An example is the code snippets found in Figure 5.4.1. Initially in the java server page a direct reference to an entity bean exists. This code is modified to make a call to the session bean in the changed code of the figure. Also, in Figure 5.4.1 note that the session bean contains a method that makes a call to the entity bean that the java server page previously accessed.



```
Enumeration enums = HelmetHome_EJB.findAll();
```

```
Enumeration  enums = SessionFacade_var.findAllHelmet();
```

Session Façade generated code calls entity bean:

```
public  Vector  findAllHelmet(){
    Vector v_toReturn = new Vector();
Enumeration enum;
Enumeration enum2;
try{
    enum = HelmetHome_var.findAll();


    while(enum.hasMoreElements()){
Helmet o =  (Helmet)PortableRemoteObject.narrow(enum.nextElement(), Helmet.class);
    v_toReturn.add( o.getValueObject() );
}

}catch(java.rmi.RemoteException re){System.out.println(re.toString());}
catch(javax.ejb.FinderException fe){System.out.println(fe.toString());}
    return v_toReturn;
}
```

Figure 5.4.1 Session Façade Code Transformation

The justification for this code being a complex refactoring is as follows:

Original Source Scope: contains java server pages, servlet files and (references to) entity beans.

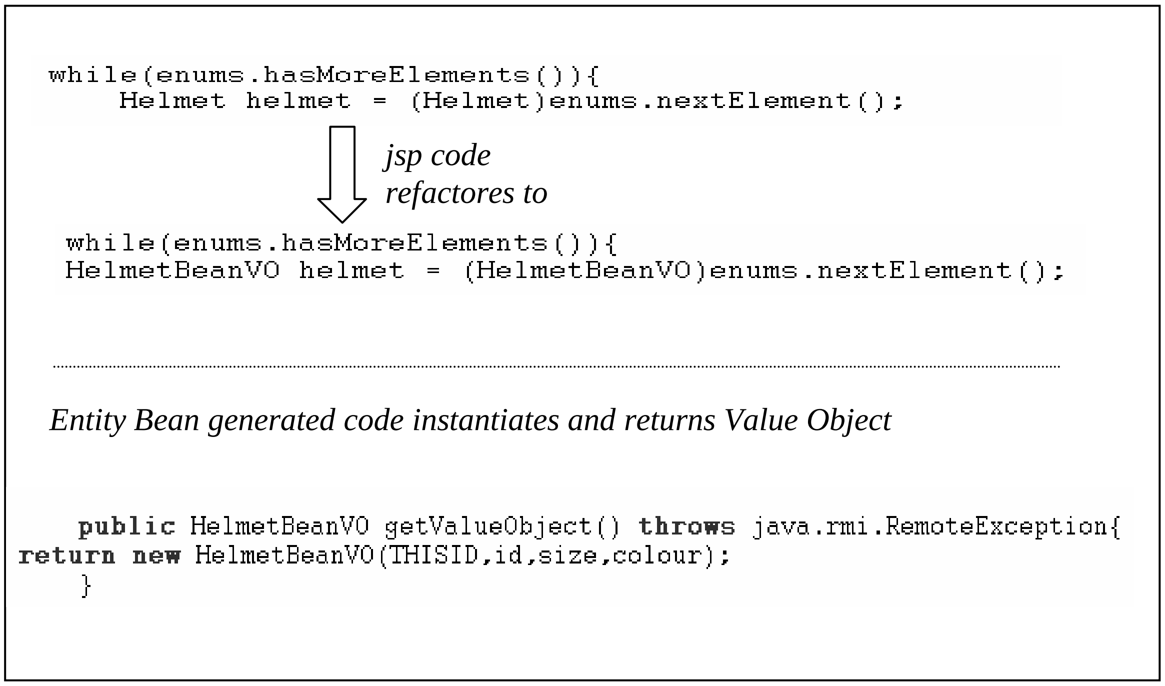
Target Source Scope: contains modified java server pages, servlet files, modified and three new classes that make up one session bean.

Nested Refactorings: The CreateSessionFacadeCRefactoring requires the use of the complex refactoring CreateSFBeanHomeReferencesCRefactoring. While it does not require any existing eclipse refactorings, the requirement of a smaller refactoring (complex or otherwise) is sufficient.

Knowledge Access: This refactoring accesses the Rule Store repeatedly to create a Session Bean, and this is shown in four import lines at the beginning of the code. This refactoring also accesses some General Development Tasks of eclipse to create ICompilationUnits, import lines, methods and package declarations. Due to these four properties, CreateSessionFacadeCRefactoring.java is an implemented complex refactoring.

CreateValueObjectCRefactoring is the second complex refactoring in the Session Façade with Value Object design patterns. It was created for the purposes of this research and according to the criteria for complex refactoring. The Original Source Scope is multiple files and multiple file types as shown by the corresponding Original Project Analysis package that accesses servlets, java server pages, the entire project structure and information from the user. The Target Source Scope is again multiple files – as will be shown by the change this complex refactoring introduces. Again, the Design Pattern Knowledge is shown in the import lines that import ProjectStructureRules.java from the Rule Store, and also in the sequences of commands in the code that will be discussed shortly. The task of this refactoring is to introduce a new Value Object into the project and modify the client and business tiers to handle the return of this value object. One value object corresponds to one entity bean and contains only private fields identical to that of the entity bean and accessor methods to these fields.

This refactoring is created with the name of the bean to which the value object conforms. It creates the value object compilation unit, gets the actual compilation unit of the bean and then copies all the fields in that entity bean over to the value object. Accessor methods for these fields are then created. Next, the refactoring creates a method in the entity bean to return the corresponding value object. A constructor for the value object is also created as is an import line in the value object. Additionally, this complex refactoring requires the modification of the java server pages and servlet pages. Wherever a line returns an actual entity bean or a vector of entity beans this is changed to return a value object or a vector of value objects, respectively, instead. This is currently done with the help of the AnalyzeJSPFile.java and AnalyzeServletFile.java. An example is found in the code snippets found in Figure 5.4.2. Initially the java server page makes a call to the entity bean (or session bean) and an entity bean is returned. After applying the complex refactoring the java server page takes a value object as the returned item. Also, note that in Figure 5.4.2 the entity bean contains a method that instantiates and returns a value object.



```
while(enums.hasMoreElements()){
    Helmet helmet = (Helmet)enums.nextElement();
```

```
while(enums.hasMoreElements()){
HelmetBeanVO helmet = (HelmetBeanVO)enums.nextElement();
```

Entity Bean generated code instantiates and returns Value Object

```
   public HelmetBeanVO getValueObject() throws java.rmi.RemoteException{
return new HelmetBeanVO(THISID,id,size,colour);
   }
```

Figure 5.4.2 Code Transformation with Value Object

The justification for this code being a complex refactoring is as follows:

Original Source Scope: contains java server pages and servlets.

Target Source Scope: contains java server pages, servlet files and entity bean modifications as well as the creation of a new class, the value object.

Nested Refactorings: this refactoring does not contain any nested refactorings.

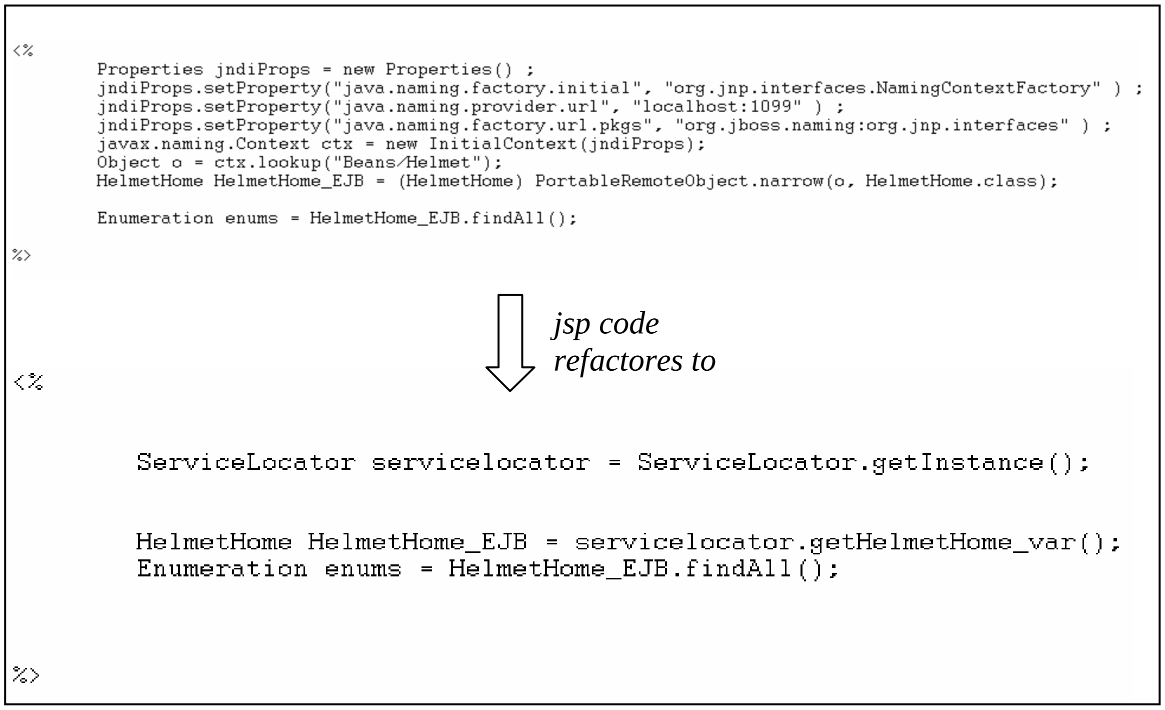
Knowledge Access: this refactoring accesses the Rule Store for some information but more importantly is the logic encompassed in the code. Only accessor methods are created on private fields, which corresponds to a value object format. Although this information is not included in the Rule Store it is nonetheless Design Pattern Knowledge. Due to these four properties, CreateValueObjectCRefactoring.java is an implemented complex refactoring.

5.4.2 Service Locator Refactoring

CreateServiceLocatorCRefactoring.java is a complex refactoring created for the purposes of this research and according to the criteria for a complex refactoring. The Original Source Scope is multiple files and multiple file types as shown by the corresponding Target Project Analysis package that accesses java server pages, servlets, the entire project structure and information from the user. The Target Source Scope is again multiple files, as will be shown by the change this complex refactoring introduces. Again the Design Pattern Knowledge is shown in the import lines that import ProjectStructureRules.java, JavaClassRules.java, ServiceLocatorClass.java and ServiceLocatorTemplate.java. The task of this refactoring is to introduce a new class that performs all lookup information for an application. This single class does the lookup once only, and maintains properties similar to that of a singleton class in that once it is created, only that instance is ever retrieved again. In this way, the lookups are performed only once, saving calls across the network.

This refactoring is created with knowledge only of the project. It creates an instance of ServiceLocatorTemplate which creates the actual ServiceLocator class with a private constructor and a static "**getInstance**" method which returns the instance of the class after it has been created.

The CreateServiceLocatorCRefactoring uses the AnalyzeJSPFile and AnalyzeServletFile to change the java server pages and servlets. Calls are made to the Service Locator to get access to the home interface of the entity beans. An example is found in Figure 5.4.3. Initially the java server page contains lookup code to access an entity bean's home interface. After applying the refactoring, one sees the call is made to the Service Locator instead.



```
<%
        Properties jndiProps = new Properties() ;
        jndiProps.setProperty("java.naming.factory.initial", "org.jnp.interfaces.NamingContextFactory" ) ;
        jndiProps.setProperty("java.naming.provider.url", "localhost:1099" ) ;
        jndiProps.setProperty("java.naming.factory.url.pkgs", "org.jboss.naming:org.jnp.interfaces" ) ;
        javax.naming.Context ctx = new InitialContext(jndiProps);
        Object o = ctx.lookup("Beans/Helmet");
        HelmetHome HelmetHome_EJB = (HelmetHome) PortableRemoteObject.narrow(o, HelmetHome.class);

        Enumeration enums = HelmetHome_EJB.findAll();

%>
```

```
<%

        ServiceLocator servicelocator = ServiceLocator.getInstance();


        HelmetHome HelmetHome_EJB = servicelocator.getHelmetHome_var();
        Enumeration enums = HelmetHome_EJB.findAll();



%>
```

Figure 5.4.3 Code Transformation with Service Locator

The justification for this code being a complex refactoring is as follows:

Original Source Scope: Contains java server pages and servlet files.

Target Source Scope: Contains java server pages and servlet file modifications as well as the creation of a new class: the ServiceLocator.

Nested Refactorings: This refactoring does not contain any nested refactorings

Knowledge Access: This refactoring accesses the Rule Store repeatedly to create a session bean and this is shown in four import lines at the beginning of the code. This refactoring also accesses some General Development Tasks of eclipse to create ICompilationUnits. Due to these four properties, CreateServiceLocatorCRefactoring.java is an implemented complex refactoring.

5.5 State of Implementation

The Design Pattern Developer wizard is implemented as a java project in eclipse. It contains one source folder **src** which contains all the packages and source code. DPD runs with the Java Development Kit 1.4.0 and has seven java archive files on the build path. They and their purposes are as follows: jdtcore.jar, jdt.jar, both used for java file manipulation; boot.jar required to include the plug-in in the launching of eclipse; resources.jar used for file manipulation; runtime.jar to launch the actual plug-in; workbench.jar and swt.jar both used for establishing the user interface of DPD. Not all of these java archive files are included in the plug-in directory when actually deploying the plug-in. The reason for this is testing purposes. The initial testing of DPD was done by launching a new instance of eclipse and the boot.jar, swt.jar and workbench.jar files need to be specified here.

5.5.1 Project Metrics

Metrics were taken for the entire project and for the Session Façade with a Value Object pattern. The latter will be the primary pattern discussed and was the pattern used in the experiment described in chapter five. The other three design patterns are smaller than the Session Façade Value Object pattern and are encompassed, to a certain extent, in this pattern.

Metrics for the Design Pattern Developer were taken using JavaNCSS [JavaNCSS03] a source measurement suit for Java, which is open source. The Design

Pattern Developer contains eighteen packages, 86 classes, 936 functions and just over 14000 non commented lines of code. The Session Façade with a Value Object pattern has 6 packages, 38 classes, 384 functions and 5374 lines of code. Each package in the Session Façade with Value Object pattern averages 6 classes per package, 10 functions per class and 140 non commented lines of code per class.

5.6 Chapter Summary

Chapter five began with a demonstration of the seven page wizard provided by the Design Pattern Developer. Using the design pattern Session Façade with Value Object as an example, the chapter showed the changes that occurred by applying the tool a J2EE application. The tool architecture was given, identifying three key packages that implement the change and one package for the user interface. The chapter then discussed in detail the complex refactorings implemented by the tool. A brief look at the state of implementation concluded the chapter.

6 AN EXPLORATIVE CASE STUDY

In order to evaluate the success of the concept of tool support for complex refactoring to design patterns, an explorative case study of the Design Pattern Developer tool was performed. The experiment ran with ethics approval from the University of Calgary and ran in the Software Engineering Lab at the university. The letter of approval from the ethics committee at the University of Calgary can be found in Appendix C. The explorative case study was performed firstly to provide initial evidence in support of support for complex refactoring to design patterns and secondly to generate feedback on the tool's user interface and functionality. What follows is an overview of the experiment based on the documents submitted for ethics approval. Following that is a discussion of what actually transpired during the experiment, a detailed analysis of the results of each participant and finally, a conclusion of the experiment.

6.1 Experiment Overview

6.1.1 Experimental Context

Research Question Addressed: Can we improve the productivity of refactoring to design patterns using the Design Pattern Developer?

Hypothesis: Using the Design Pattern Developer Tool improves the productivity of refactoring to design patterns, where productivity is represented by time in minutes to successfully refactor to a design pattern, and success is measured by 100% passing of all test drivers in the provided Java 2 Enterprise Edition applications. That is for productivity variable P_{-DPD}, (without using the Design Pattern Developer), and productivity variable P_{DPD} (with the Design Pattern Developer),

$$P_{DPD} \text{ -> } P_{-DPD}$$

where P = time *t* to successfully refactor to a design pattern

and success *s* = 100% passing of all test drivers.

Background: Existing tool support for refactoring does not support refactoring to design patterns. A developer is forced to perform multiple small refactorings and keep track of what is occurring through the changes. The Design Pattern Developer Tool examined compiles small refactorings required to refactor to a design pattern as well as knowledge of an initial J2EE application, for a developer.

6.1.2 Experimental Design

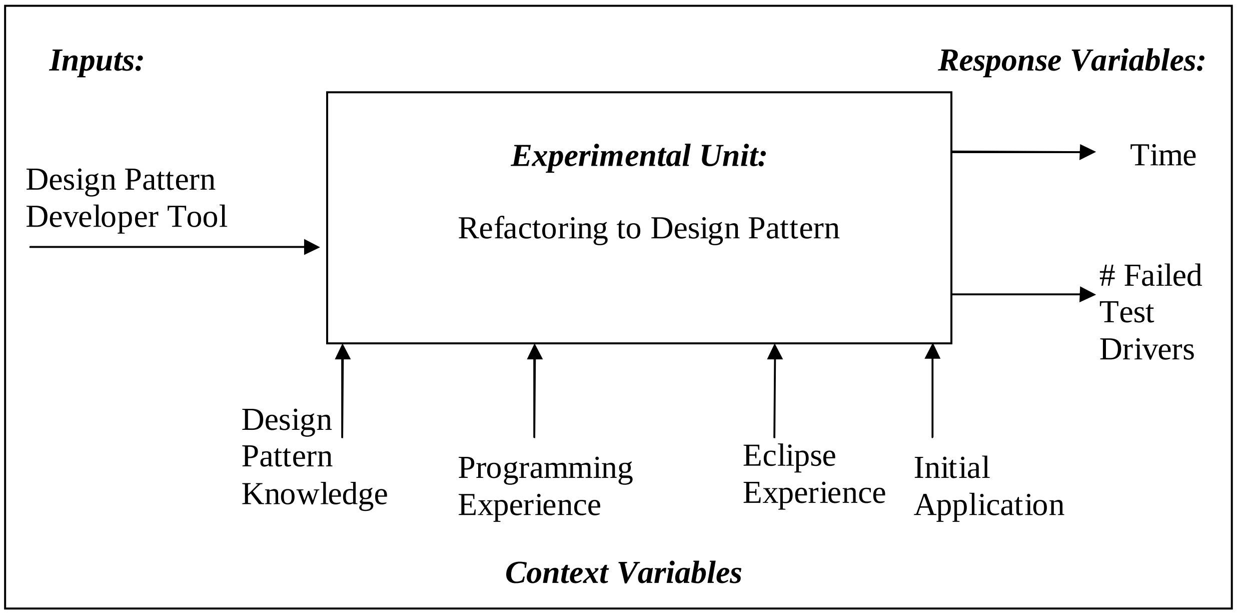
<u>Population from which Subjects are Drawn:</u> The goal was to have 12 subjects taken from a fourth year computer science course and/or from Software Engineering graduate courses. What actually occurred during the study is discussed in Section 6.2. The explorative case study asked for volunteers from this domain and advertised that food would be given after the experiment. As well, the volunteers would be paid $10 per hour of their time. In reality, nine subjects attended the case study and were given pizza and $20 each for the two hours that the experiment ran.

<u>Process by which the Subjects are Assigned to Treatments:</u> The initial plan was to have the subjects randomly draw a piece of paper to indicate if they would start with the tool or without when refactoring to design patterns.

Student (randomly selected)	Application_1	Application_2
1	Tool	No Tool
2	No Tool	Tool
. . .	. . .	. . .
7	Tool	No Tool
8	No Tool	Tool

<u>Experimental Unit:</u>

Figure 6.1.1 shows the inputs, outputs and context variables for this experiment. They are then described below.



Inputs:	Design Pattern Developer Tool (use of / no use of)
Response Variables:	Time represented in minutes
	# of failed test drivers using Junit test framework [JUnit03]
Context Variables:	Developer (Experimental Subject) Design Pattern Knowledge
	Developer (Experimental Subject) Programming Experience
	Developer (Experimental Subject) Experience with Development Environment, Eclipse
	Initial Application

Context Variable	**Description**
Developer Design Pattern Knowledge	all experimental subjects had an in class introduction to J2EE design patterns or some work experience before they participated in the study
Developer Programming Experience	since experimental subjects are at a minimum 4th year university students, they had at least 2 years part time experience with Java.
Experience with	Experimental subjects have experience working in eclipse since

Development Environment	it is introduced to them in the undergraduate and graduate courses.
Initial Application	there were be 2 initial applications, both Java 2 Enterprise Edition applications, both containing the same number of servlets, jsps and entity beans, and both with the same structure.

<u>Experiment Steps</u>

Step 1: General introductions. The plan was for subjects to randomly draw a piece of paper to indicate whether or not they began refactoring with the Design Pattern Developer Tool. The subjects would be split into 2 groups. Group A would begin without the tool, Group B would begin with the tool.

Step 2: Group A would be asked to refactor to the Session Façade with a nested Value Object design pattern. The subjects would be asked to work individually, would start at the same time, and would be asked to write down the time listed on the computer clock. If the subject takes a break s/he would be asked to record the time they stopped at and the time they began again. Introduction of these rules would be given and questions related to design patterns would be answered.

Step 3: Group B would be given an introduction demo to the tool. A question period would be provided.

Step 4: They would be asked to refactor Application_B, using the Design Pattern Developer Tool to refactor to the Session Façade nested Value Object pattern. The subject would be asked to work individually, and would be given up to 1/2 hour to complete the task. They would all start at the same time, and would be asked to write down the time listed on the computer clock. If the subject takes a break (and lunch) s/he would be asked to record the time they stopped at and the time they began again. Introduction of these rules would be given and questions related to design patterns would be answered.

Step 5: The groups would be switched and asked to do the task they have not yet performed – either Step 3 or Step 4&5, depending on which task they initially completed.

Refactorings:	Create 1 Session Façade
	Create 1 Value Object corresponding to only 1 Entity Bean
	Make required changes in 2 corresponding jsp and 2 servlet files.

The refactorings performed were to create one session façade, create one value object corresponding to only one entity bean and finally, make the required changes in two corresponding .jsp files and two servlet files. The outcome measures were firstly the time required to accomplish the task and secondly the success of the test drivers.

6.1.3 Data Collection

Entity:	Development Time	Entity:	Logic Errors
Attribute:	Actual Time	Attribute:	Test Drivers
Unit:	Minutes	Unit:	# Tests Passed

6.2 Summary of Events

The experiment was held in one software engineering lab. Nine participants volunteered, of varying skill levels in Java programming, but all with at least beginner knowledge of entity beans and J2EE design patterns. The students were randomly split into two groups based solely on the location in which they sat when they entered the room. Four participants at the back of the lab became Group B, the group that started by manually refactoring to design patterns. The remaining five at the front of the room became Group A, who started by refactoring with the Design Pattern Developer tool.

A small introduction was given, describing the purpose of the experiment and the general tasks the participants were required to perform. The participants were given the handout found in Appendix C. Group B started their task immediately. Group A listened and watched a demonstration of the tool and then was asked to use the tool to refactor. Both groups refactored to the Session Façade with a Value Object pattern. The projects used in the experiment are described in Section 6.3.

The demonstration, with questions, lasted twenty minutes and then Group A began their task. Some questions arose concerning entity beans and deployment and these were answered accordingly. After a half an hour all participants were asked to stop the work they were doing, regardless of whether or not they had completed their task. The participants were given ten minutes to fill out the questionnaire provided and take a biological break.

Next the groups were switched. Group A began immediately manually refactoring to the Session Façade with Value Object pattern. Group B watched the same demonstration as Group A had watched, and was given a chance to ask questions. This again took twenty minutes. Group B then began refactoring to the design pattern using the tool. Again questions concerning J2EE development were answered. After a half hour all participants were asked to stop what they were doing regardless of where they were at in the experiment. The participants were given ten minutes to finish filling out the questionnaire, and pizza and payment for the participants' time, was provided.

6.3 Projects

Two J2EE applications were created, on which the tool was applied and to be used for the experiment. The first project was called RiverKayaking. It consists of six entity beans, twelve servlets and twelve java server pages. The participants applied the tool to this project to manually refactor to the design pattern. The ShoppingNetwork was the second project the participants were asked to manually refactor. It consisted of four entity beans, eight servlets and eight java server pages. When asked to refactor, the participants were asked to refactor only one entity bean, for simplicity's sake. This meant changing two java server pages and two servlets. The two projects were the same in this respect.

6.4 Participant Results

Nine people participated in the experiment. Table 6.4.1 details the results of each participant's activities. The 1st and 3rd column show the time spent working on each task type of refactoring (with the tool and without, respectively). Columns 2 and 4 show whether or not they completed tasks found in columns 1 and 3 respectively. Columns 5 and 6 show the amount of experience they have with J2EE applications and design patterns. Finally, Column 7 shows whether or not they would choose to use the tool over manually refactoring.

	Context Variables						
Participant	Tool Time	Completed (Test Drivers Passed)	Manual Time	Completed (Test Drivers Passed)	J2EE Experience	Design Pattern Experience	Tool over Manual
1	7	Yes	47	No	1 part time month	1 part time year	Yes
2	Did Not Complete Experiment						
3	24	Yes	48	No	3 years	4+ years	Yes
4	17	Yes	60	No	4 years	N/A	No
5	14	Yes	33	Yes	2 years	0	Maybe
6	7	Yes	N/A	No	< 1 year	1 year	Yes
7	17	Yes	35	No	1.5 years	6 months -1 year	Maybe
8	34	No	27	Yes	3 years	3 years	No
9	19	Yes	N/A	No	2.5 years	1 year	Yes

Table 6.4.1 Experiment Results

Participant #8 was the only person who did not finish with the tool. This subject found a bug in the tool that caused him to spend time correcting the application before he could test the refactoring.

Participant #5 and #7 who both answered “Maybe” to using the tool over manually refactoring, both elaborated that a combination of the two would be an alternative. Question #3 from the Post Experiment Questionnaire asks, “If you had a choice between using a tool like DPD and manually refactoring, which would you choose and why?” (Please see Appendix C for the entire questionnaire). Participant #7 answered the following to this question: “A tool as long as it also generates [a] JUnit stub class which fails by default. This will force me to write the test drivers to assure quality of refactored code. I must say I do not rely totally on any tools for complex refactoring. There [is] still need for human intervention.” Participant #5 answered the following to this same question: “[It] would depend on [the] situation. Probably use the tool and then manually adjust afterwards." Question #4 asks whether or not the participant would like to see further implementation of the tool. Despite the "Maybe" answer on question #3, Participant #5 answered to question #4, "Yes. Things like this are good for such complex tasks."

Participants #4 and #8 both said they would rather manually refactor than use the tool. Participant #8, in response to question #3 said the tool is not advanced enough yet and doing it manually yields the opportunity to review code. Still, Participant #8's response to seeing further implementation of the tool was yes.

Participant #4 answered the following: "Manual. I come from the 'code crafter' school rather than the 'test and refactor' school. So, chances are I wouldn't need it on good code, but on bad code I'd worry about safety. (If the code is bad, how can you trust the test?)".

While these two results are not soundly in favour of the use of the Design Pattern Developer, one must recognize two points. Participant #8 is still willing to see future development and growth in such a tool. Thus, support for the idea is still present. The second point is that Participant #4's description of coding leans away from agile practices and the Design Pattern Developer and complex refactoring are both based on agile concepts. Despite these two comments, one cannot ignore that 6 out of 8 participants that saw the study through to completion are in full support of such a concept and further development of the tool.

The time Participant #8 records as having spent using the tool to refactor to design patterns exceeds the allotted time by four minutes. Similarly, the time that Participant #4 spent manually refactoring to the design pattern exceeds the allotted time. It is thought that these individuals recorded the times on their computer later than requested, that is, after they were asked to stop their work. Regardless, neither of these individuals finished their task in the extra few minutes taken which is not a positive sign in either situation.

6.5 Statistical Analysis Using Paired T-Test

To give weight to the claims made above, statistical analysis of the mean time to refactor in each situation was analyzed. A Paired T Test was performed as per [Milton95, 354]. The Paired T-Test is suitable for problems where two random samples are taken but are not independent of each other. Each observation (or subject) in one of the samples is paired with an observation in the other sample, either naturally or by design. [Milton95, 353]. In this situation, each observation, or subject performing the refactoring with the tool, was directly paired with his/her corresponding performance without the tool. The two samples are not independent of each other, by design. The Paired T-Test shows the differences between samples that are related to each other and was therefore appropriate for this analysis. No assumption is made on the sample size necessary for a Paired t-test, as found in [Milton95, 353].

The hypothesis for this experiment, as previously stated, was that using the Design Pattern Developer Tool improves the productivity of refactoring to design patterns, where productivity is represented by time in minutes to successfully refactor to a design pattern, and success is measured by 100% passing of all test drivers. For statistical analysis, the mean time (in minutes) to refactor to the Session Façade Value Object design pattern with the tool was compared with manually refactoring to the same pattern. Here just the time to refactor to a design pattern is addressed, success is combined with this analysis later. Thus, the research (or alternative) hypothesis is that $\mu_X < \mu_Y$, where μ is the mean time to refactor, X is the group that refactored with the tool and Y is the group that refactored without the tool. The null hypothesis is that that $\mu_X = \mu_Y$, or $\mu_D = 0$. Here, μ, X and Y are as above, and D represents the mean differences between populations X and Y. μ_D is the mean of the population of differences and $\mu_D = 0$ is equivalent to the null hypothesis $\mu_X = \mu_Y$. The goal is to prove these points to be false. The value of the test statistic, as per [Milton95, 355] is

$$\frac{D-0}{S_d / \sqrt{n}}$$

Here, D is the sample mean of the sample of difference scores, S_d is the sample standard deviation of the sample of difference scores and n is the number of subjects concerned. Using columns 1 and 3 from Table 6.4.1 and functionality provided by Excel, the following data was acquired.

t-Test: Paired Two Sample for Means		
	X	Y
Mean	19.66666667	41.66666667
Variance	119.8666667	147.8666667
Observations	6	6
Hypothesized Mean Difference	0	
Degrees of Freedom	5	
t Statistic	-2.728764161	

Table 6.5.1 Paired t-Test to Compare Refactoring With and Without the Tool

X represents the tool refactoring scenario, Y represents the manual refactoring scenario. The mean shows the average time required to perform each task. The variance shows the difference between the variable and its mean [Milton95, 54]. Only 6 subjects were used in observing because only 6 provided sufficient data to perform this test. The hypothesized mean difference represents the null hypothesis, that there is no difference between the two test statistics taken. Degrees of freedom is computed as the number of observations minus 1. The t Statistic is computed using the formula from above. Once the t value has been determined, a Table of Significance, such as the one found in [McGraw, 732] is consulted. Working with an alpha value of .05, it is determined that the absolute value of the computed t value of -2.728764161 is greater than the T value provided in the T distribution, which is 2.2.015. Thus the null hypothesis $\mu_X = \mu_Y$ with an alpha value of 0.5 is false. Since the mean of the X group is less than the mean of the Y group, $\mu_X > \mu_Y$ is not possible. Thus, the alternative hypothesis is true: $\mu_X > \mu_Y$. That is, the time to refactor to the design pattern using the tool is significantly less than the time to refactor to the design pattern without the tool, using an alpha value of .05, the generally accepted alpha value found in [Milton95].

By examining the YES/NO results of the success each method achieved, it is found that 2 out of 6 participants were successful in manually refactoring to a design pattern while 5 out of 6 participants were successful in using the tool to refactor to design patterns. Combining this with the success of the alternative hypothesis from above it is concluded that the time to refactor to the design pattern using the tool is significantly less than the time to refactor to the design pattern without the tool and more participants successfully refactored with the tool than without.

Using the tool to refactor to design patterns, the participants sat through a twenty minute introduction to the tool. By applying the Paired t-test to these to groups, where X is without the introduction and Y is with the introduction to the tool, one finds the mean time to not be significantly different between the two situations. When applying the paired t-test, the value cannot be computed because a zero appears in the quotient in the formula. The learning curve of using such a tool does not have a significant impact on the time required to perform the activity.

Participant	Tool Time	With Introduction
1	7	27
2	N/A	N/A
3	24	44
4	17	37
5	14	34
6	7	27
7	17	37
8	39	59

Table 6.5.2 Tool Refactoring Time with Introductory Lesson

6.6 Enhancements Based on Qualitative Feedback

The consistent comment made among all participants concerned two items. First, the tool needs to include modification of the deployment descriptors. The deployment descriptors occupied much time when actually deploying the project and it would be much more useful if the tool could modify these files. Conceptually, this involves encompassing more J2EE knowledge into the tool. Second, the tool needs to include the generation and/or modification of unit tests in order to be of more use to a developer.

6.7 Limitations of the Study

Issues that threaten the validity of the study must be recognized. They are as follows:

- the number of participants in the study was small. The initial goal was to have 12 students participating, but the final data includes 8 participants and the statistical analysis incorporates 6 of these. Thus, the impact that these numbers have in proving the hypothesis is recognizably small.
- many of the participants knew the person conducting the study. Consequently, answers to qualitative questions could have been influenced by a friendship or positive working relationship between the subject and the experimenter. For the same reason, the fact that the subjects were being paid must be recognized as a threat to the validity of the study.
- the amount of experience each experimental subject had with design patterns, J2EE applications and eclipse was classified as “beginner” but no concrete analysis was done on the variances of this “beginner” experience and how it affected the performance of each experimental subject.
- data on experimental subject experience with eclipse was not taken
- the subjects were not given time to learn and understand the projects they were refactoring. In a realistic refactoring to design patterns situation, the developer performing the refactoring would have knowledge of the structure of the project s/he was working with and would know what aspects of the code needed to be changed. In the study, when subjects were manually refactoring to the design pattern, they had to first read the code and get a conceptual idea of what was happening, before they changed the code. This, no doubt, cost them time. To a certain extent, the naming of the files as found in Figure 5.3.1 and Figure 5.3.2 assisted in this process, but actual quantitative data to show this learning curve was not collected.

6.8 Chapter Summary

This explorative case study was performed to yield support in favour of refactoring to design patterns, but also for academic reasons, specifically to gain experience with empirical analysis. The study was performed with ethics approval from the University of Calgary and involved 8 subjects randomly split into two groups. One group began manually refactoring to the Session Façade with Value Object design pattern and the other group watched a demo of the Design Pattern Developer Tool and then used the tool to refactor to the Session Façade with Value Object pattern. The two groups recorded the time it took to perform their tasks, then they switched tasks. Results find that the tool requires further development to incorporate more J2EE knowledge, and requires some bug fixes, but overall opinion was in support of such a tool. Statistical analysis shows a significant improvement in time to refactor to the design pattern using the tool. Limitations of the study are recognized as the size of the experiment, the familiarity with the experimenter and economic provisions made for subject participation. The last limitation of the study is a lack of analysis concerning the time required for the developer manually refactoring to gain a conceptual understanding of the application s/he was refactoring.

7 CONCLUSION

There exists a notion of emerging design in the agile world, where the design of an application evolves into a final design through numerous refactorings and added functionality resulting from new and changing requirements. In an attempt to “get the job done” design and coding are done continuously and are based on the YAGNI principle [YAGNI03]. This approach to design argues with traditional software development approaches to design. Traditional software development often implements upfront design. In an attempt to remain flexible, design patterns are used during upfront design. The research question addressed in this thesis thus is, can one bridge the gap between the YAGNI principle used in agile methodologies and provided flexibility used in traditional methodologies by supporting emerging design to design patterns? The ability to better specify what refactoring does and to provide tool support for refactorings that introduce sound design via design patterns, is a key weapon in the challenge to support emerging design to design patterns.

The increased popularity of Agile methodologies and the abundance of refactorings has resulted in various forms of support to apply these refactorings. From catalogues to tools a developer has seemingly endless leads to refactorings. Specific refactorings have different levels of complexity in terms of what they change in a given application. Support for refactorings also varies from highly available to completely unavailable.

The benefits of categorizing refactorings are many. Enhanced organization in the wake of over one hundred published refactorings is one. Increased potential to understand a refactoring to the extent that it becomes useful to a developer on a daily basis, is another. Finally, implementation of tool support for refactorings that produce significant design changes in an application, is the key motivation for solidifying the current state of refactoring.

In examining over 100 published refactorings of varying complexity, four criteria were established to classify a refactoring. The scope of the original source identified the size of the code to be refactored. The scope of the target source identified the amount of new code created. Any nested refactorings were identified in the examined refactoring. Finally, the amount and type of knowledge the refactoring required was identified as being one Abstract Syntax Tree, multiple Abstract Syntax Trees and/or knowledge of design patterns. Based on these criteria the complexity of a refactoring was defined as being one of three categories.

Atomic refactorings deal with only one Abstract Syntax Tree and tool support is abundant. Sequential refactorings deal with multiple Abstract Syntax Trees and tool support is somewhat available. Complex refactorings deal with multiple Abstract Syntax Trees and incorporate knowledge of design patterns to refactor an application to a design pattern. Tool support for such refactorings is unavailable and is the focus of this research.

Tool support to manipulate refactorings, resulted in the development of a plug-in that provides tool support for complex refactoring to design patterns. The plug-in provides support to refactor J2EE applications to one of four J2EE design patterns. It maintains three key packages that provide the support. The Original Project Analysis sets up the project for refactoring, the Target Project Analysis holds the complex refactorings that implement design patterns. The Rule Store holds the knowledge of design patterns and the J2EE domain knowledge that the complex refactorings require to implement their task. To evaluate the tool, eight subjects manually refactored a J2EE application to a specific design pattern and used the tool to refactor to that same design pattern. Analysis of the time required to refactor and degree of success was taken in each situation. While the tool has some bugs, support of the tool and the concept of refactoring to design patterns was abundant in the experiment.

Upon examination of refactoring and providing tool support for complex refactoring to design patterns, one finds capability to group refactorings by complexity for the purposes of implementing tool support to assist in emerging design. Thus as per the goals found in section 1.3, the novel aspects of this research are:

a. new categorizations of refactoring based on three different levels of complexity and knowledge
b. introduction and description of complex refactorings
c. requirements for a tool to support complex refactoring to design patterns
d. proof of the concept of tool support for complex refactorings to design patterns
e. empirical results in favour of such a tool.

Future work in this area includes:

a. complex refactorings enhancements

b. enhancing knowledge representation of design patterns
c. user interface enhancements

Proposed Enhancements

The tool described is a proof of the concept of complex refactoring to design patterns. It is not an "industrial strength" tool and there are some enhancements that need to be made. These enhancements fall into three categories.

Complex Refactorings

The complex refactorings implemented in the tool function well but could use some modifications. Firstly, it is desirable to pull out all design pattern knowledge nested in the logic of the code and move it into the Rule Store to reduce coupling between the Target Project Analysis package and the Rule Store. An example of this is in the creation of a value object where only accessor methods are allowed to be added to the value object compilation unit. This property should be moved to the Rule Store. Secondly, it is desirable to implement support to undo these complex refactorings. Lastly, including these refactorings in the existing eclipse refactoring base would ease reuse of the refactorings.

Knowledge Representation

A prime area for future work is in examining the way the design pattern knowledge is encoded in the tool. The first aspect of this knowledge is to extract it out to its own package. Another area for improvement is in the representation of the information. At the time of writing, the knowledge is hard-coded in the Rule Store. A better option would be to use descriptors to specify the structure of the given types of files.

User Interface

An apparent weakness with this tool is the user interface. Help menus need to be added to each page and general enhancements of the overall look and feel need to be made.

Experimentation

Further experimentation is needed. Firstly experimentation involving more subject, possibly in an industry setting, is desirable. Evaluating the flexibility of the application on varying J2EE applications is also desirable.

REFERENCES

[Aho et al.98] Aho, Alfred, Sethi, Ravi, Ullman, Jeffrey; Compilers Principles, Techniques and Tools; Addison-Wesley Publishing Company March 1998

[Alur et al.01] 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

[Ant03] Ant http://ant.apache.org/ (Last Visited June 27, 2003)

[Beck00] Beck, K.; Extreme Programming Explained: Embrace Change; Addison Wesley Upper Saddle River NJ, 2000; p.103-115

[Boehm85] Boehm, B.W. A Spiral Model of Software Development and Enhancement. Proceedings of an International Workshop on Software Process and Software Environments, Coto de Caza, Trabuco Canyon, California, March 27-29, 1985

[Brown03] Brown, K.; Rules and Patterns for Session Facades; IBM WebSphere Services; http://www7b.boulder.ibm.com/wsdd/library/techarticles/0106_brown/sessionfacades.html (Last Visited June 27, 2003)

[Cinneide00] Cinneide, M.; Automated Refactoring to Introduce Design Patterns; Proceedings of the 22nd International Conference on Software Engineering 2000;

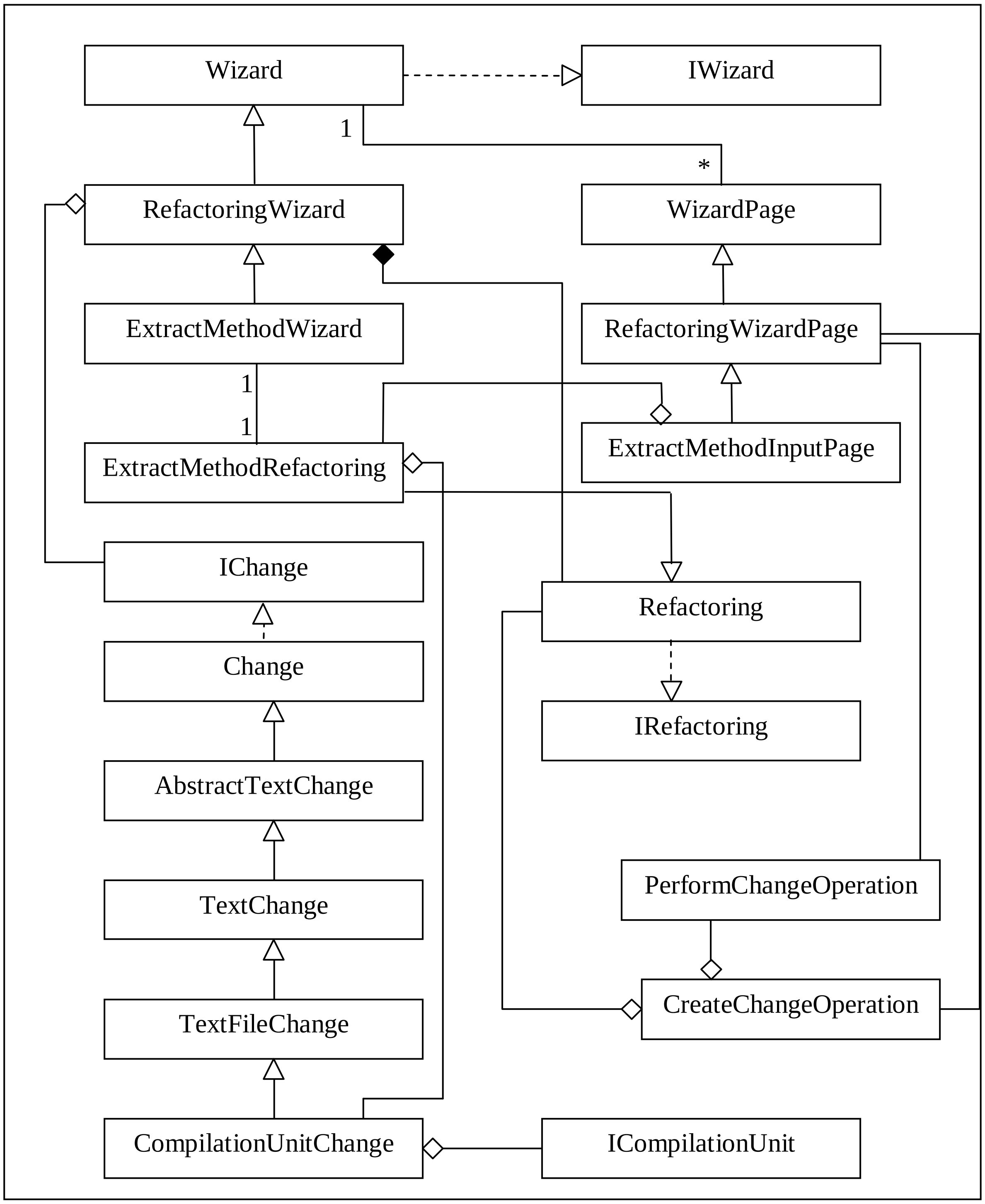
[Cockburn03] Cockburn, A., Highsmith, J., Jones, M., Crystal Main Foyer 2002 http://www.crystalmethodologies.org (Last Visited June 27, 2003)

[CVS03]	Concurrent Versioning System http://www.cvshome.org/ (Last Visited June 27, 2003)
[Ducasse et al.99]	Ducasse, S., Tieger, M., Golomingi G.; Tool Support for Refactoring Duplicated OO Code; Position Paper Software Composition Group, April 1999
[Eclipse03]	Eclipse www.eclipse.org (Last Visited May 12, 2003)
[Fowler00]	Fowler, M.; Refactoring: Improving the Design of Existing Code; Addison-Wesley ; Upper Saddle River, NJ 2000;
[GoF95]	Gamma E. Helm R. Johnson R. Vlissides J; Design Patterns – Elements of Reusable Object Oriented Software; Addison Wesley; Reading MA, 1995;
[Grand02]	Grand, Mark; Java Enterprise Design Patterns, Patterns in Java 3; John Wiley & Sons Inc. New York 2002
[IntelliJ03]	IntelliJ IDEA www.intellij.com/idea (Last Visited June 27, 2003)
[JavaNCSS03]	JavaNCSS http://www.kclee.com/clemens/java/javancss/ (Last Visited June 27, 2003)
[J2EE00]	Java Server Programming J2EE Edition; Wrox Press, Birmingham 2000
[JUnit03]	JUnit Testing http://www.junit.org/index.htm (Last Visited June 27, 2003)

[Kang03]	Kang, A.; J2EE Clustering, Part 1; JavaWorld www.javaworld.com/javaworld/jw-02-2001/jw-0223-extremescale_p.html (Last Visited June 27, 2003)
[Kerievsky03]	Kerievsky, Joshua; Refactoring to Patterns; Industrial Logic www.industriallogic.com/papers/rtp015.pdf (Last Visited June 27, 2003)
[Korman98]	Korman, Walter, F.; Elbereth: Tool Support for Refactoring Java Programs; University of California, San Diego, 1998
[Milton95]	Milton J.S.; Arnold J.C.; Introduction to Probability and Statistics: Principles and Applications for Engineering and the Computing Sciences Third Edition; McGraw-Hill Series in Probability and Statistics, McGraw-Hill Inc. New Yorkk NY 1995.
[Opdyke92]	Opdyke W.; Refactoring Object-Oriented Frameworks; PhD Dissertation, University of Illinois at Urbana-Champaign 1992
[Roberts99]	Roberts, Donald B.; Practical Analysis for Refactoring; PhD Dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, 1999
[Roberts et al.99]	Roberts, D., Brant, J., Johnson, R.; A Refactoring Tool for Smalltalk; Department of Computer Science, University of Illinois at Urbana-Champaign
[Schwaber02]	Schwaber, K., Beedle, M.; Agile Software Development with Scrum; Prentice Hall, Upper Saddle River, NJ 2002

[Tokuda99]	Tokuda, Lance A.; Evolving Object-Oriented Designs with Refactorings; The University of Texas at Austin, 1999
[vanVillet93]	van Villet, Hans; Software Engineering Principles and Practice; Vrije Universiteit, Amsterdam; John Wiley & Sons Ltd 1993
[YAGNI03]	YAGNI http://xp.c2.com/YouArentGonnaNeedIt.html (Last Visited June 27, 2003)

APPENDIX A



Partial Class Diagram of Eclipse’s Refactoring Support using Extract Method

J2EE Applications Overview

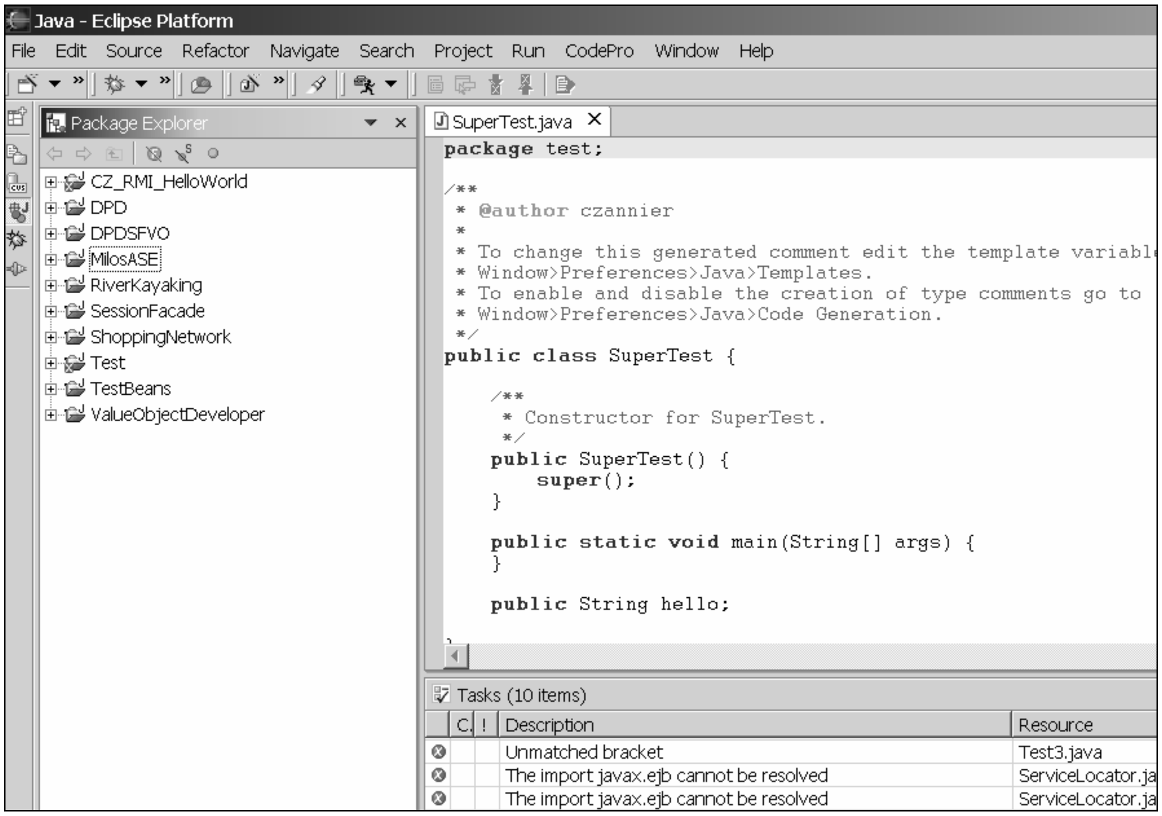
Before delving into the new aspect of refactoring, a brief description of the chosen domain must be given. Java 2 Enterprise Edition applications are becoming increasingly popular and are the focus domain of this research [J2EE00]. The basic structure of a J2EE application is a three tier structure with a database back-end connected to the client tier (Java Server Pages, Html pages) via Enterprise Java Beans [J2EE00]. Due to the client-server architecture, many issues arise when developing these applications, the major one being heavy network traffic resulting in poor performance of various J2EE applications. The J2EE pattern catalogue, [Alur et al.01], specifies some bad practices and refactorings that are encountered when developing J2EE applications. These bad practices are separated among the three inherent tiers in J2EE applications: presentation, business and integration tiers. The chosen area of focus for this research lies in the business and integration tiers, in the hopes of improving network latency.

Some J2EE design issues are as follows: mapping the object model directly to the entity model generates many entity beans, often more than necessary, and the network overhead increases with accessing an increasing number of entity beans. Similarly, one should not map the relational model directly to the entity model. Here the problem though, is scalability, not network overhead. When one wants to add to the application, a change in one will directly affect a change in the other. Another bad practice at the business tier is exposing all enterprise bean attributes with getter and setter methods. Numerous method calls are made across the network just to receive all the data in the entity bean. Placing service lookup information in each client forces numerous Java Naming Directory Interface (JNDI) lookups and introduces unnecessary code into numerous files. Network overhead increases as does coupling between the client and the integration tier. These last two bad practices will become particularly interesting as we explain the proof of concept in chapter five. Increased network overhead is also introduced by using entity beans as small fine grained components rather than more complex, coarse-grained components. Another bad practice at the business tier is using entity bean finder methods to return a large results set. This results in numerous remote calls and much network overhead.

APPENDIX B

The Eclipse Environment

Eclipse is an open source integrated development environment produced by IBM. A previously stated, it was chosen as the development environment for this research due to its availability of code and plug-in development capabilities. Eclipse is made up of perspectives, concurrent versioning system capabilities, ant build capabilities and the like. Perspectives consist of various windows and menus for application development. A prime example is the Java perspective which provides a java editor window, a task (compile error) window and a project hierarchy window showing packages, classes, methods and fields. A screen shot is shown in below.



```
package test;

/**
 * @author czannier
 *
 * To change this generated comment edit the template variabl
 * Window>Preferences>Java>Templates.
 * To enable and disable the creation of type comments go to
 * Window>Preferences>Java>Code Generation.
 */
public class SuperTest {

    /**
     * Constructor for SuperTest.
     */
    public SuperTest() {
        super();
    }

    public static void main(String[] args) {
    }

    public String hello;
```

C	!	Description	Resource
		Unmatched bracket	Test3.java
		The import javax.ejb cannot be resolved	ServiceLocator.ja
		The import javax.ejb cannot be resolved	ServiceLocator.ja

The Eclipse 2.0 Java Perspective

Eclipse runs via a series of plug-ins that establish its capabilities upon launch. Any new plug-in developed for eclipse beyond its base set of plug-ins is placed in the $ECLIPSE_HOME/plugins directory. A new plug-in requires a Java Archive file and a plugin.xml file to load the plug-in when eclipse is launched. These are both placed in the specific plug-in's folder which is then placed in the eclipse plug-in directory. For the Design Pattern Developer, the tool created for this research, the plug-in folder DPD contains a plugin.xml file, a DPD.jar file and three Java Archive files required for the plug-in during runtime. The DPD.jar file contains all classes of the plug-in. The plug-in can be tested in a new instance of eclipse without placing the DPD file into the plug-in directory, but the true test of a plug-in's functionality is in launching one instance of eclipse with the plug-in installed in the $ECLIPSE_HOME/plugins directory.

Design Pattern Developer Plug-In

The Design Pattern Developer was created with the Plug-in Development Environment contained in eclipse. The Plug-in Development Environment wizard creates an extension point into which a plug-in plugs. For the Design Pattern Developer the extension point used was a wizard, which is how DPD was implemented in eclipse. The plug-in development environment created two classes: DPDPlugin.java and DPDWizard.java to launch the plug-in and to launch the wizard, respectively. DPDWizard.java contains all the pages necessary to run the plug-in.

plugin.xml

The plugin.xml file is also generated by the plug-in development environment, but can be modified by the developer. It specifies where to plug in to, and what files are required at runtime. It contains four sections.

<ins>&lt;plug-in></ins>

This is the global tag that sets all global attributes for the plug-in. An id, name, version and provider-name are provided, and are modifiable by the developer. The key attribute provided in the plug-in tag is the class. Here, DPD.DPDPlugin is specified as the location and file to be launched when the plug-in is loaded into eclipse.

<ins>&lt;runtime&gt;</ins>

The runtime tag specifies where to look for the class files. The DPD plug-in specifies DPD.jar, the java archive file containing all the classes for the plug-in. Note that this is the DPD.jar file contained in the DPD folder, which is placed in the $ECLIPSE_HOME/plugins directory.

<ins>&lt;requires&gt;</ins>

The requires tag specifies all Java archive files that should be included in the plug-in. The plugin.xml file for DPD requires six eclipse plug-ins: org.eclipse.core.resources, org.eclipse.ui, org.eclipse.core.runtime, org.eclipse.jdt, org.eclipse.jdt.core, and org.eclipse.jdt.ui. The Java Development Tools packages (packages with "jdt" in them, e.g. the last three listed) are required by any application or plug-in doing Java development and yields access to Java elements such as ICompilationUnit, discussed in section 3.3.3b. The package org.eclipse.core.runtime is used to launch the plug-in. The package org.eclipse.ui establishes a window in a perspective. Finally, org.eclipse.core.resources yields access to the actual files for file manipulation and modification.

<u><extension></u>

The extension tag lists all extension points to which the plug-in connects. An extension point, for example, could be a view or an action to create a small window or button in an eclipse perspective, respectively. The plugin.xml file used in DPD specifies org.eclipse.ui.newWizards as its extension point because the plug-in is implemented as a wizard. Finally, the plugin.xml file specifies the class to be launched – in this case it is DPD.wizards.DPDWizard.

In eclipse's plug-in development environment one can specify these tags of the plug-in.xml file through the click of a button on the Plug-in Manifest Editor, or by typing the code manually.

Ant Build File

After the plugin.xml file is created a build file is created to create the plug-in directory for DPD. An ant build file contains three targets that do this automatically.

<u>Init</u>

The init target initializes all global variables, establishing which directory needs to be examined and compiled and where to put the compiled classes.

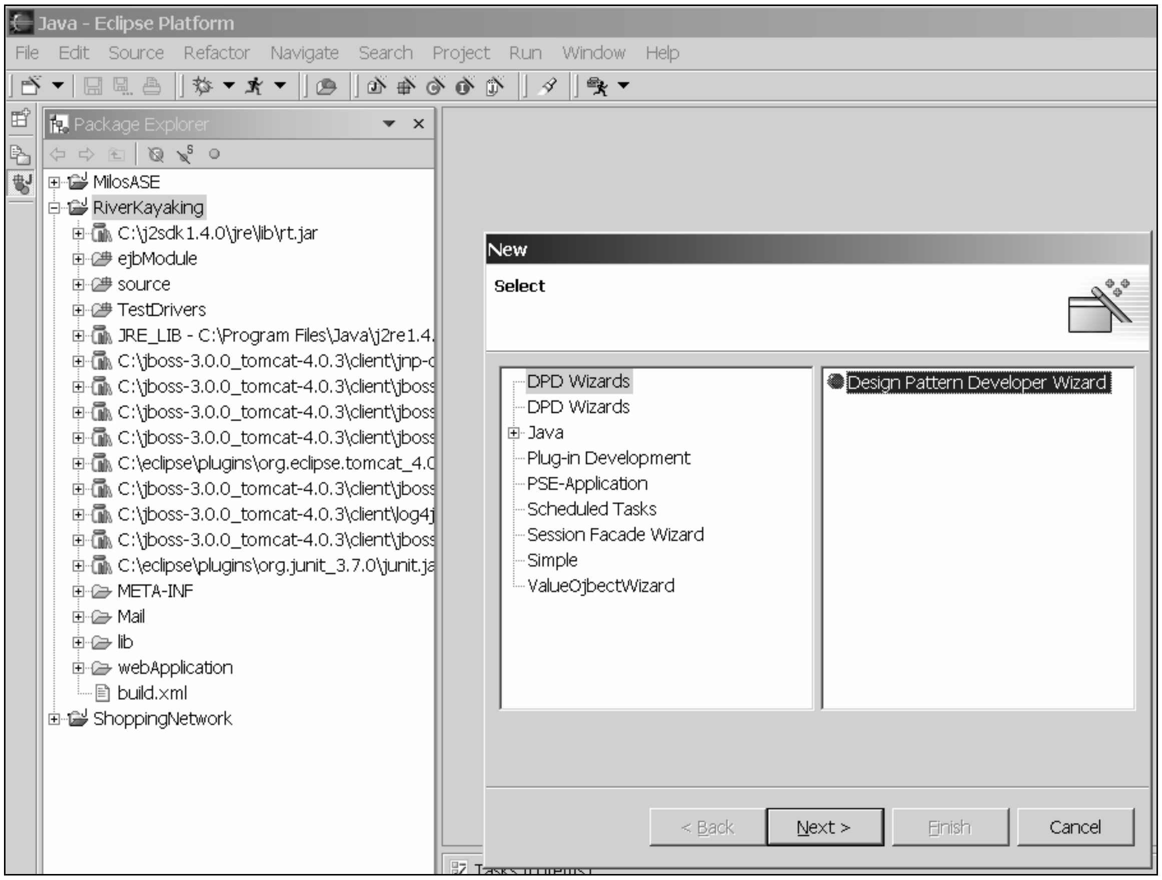
<u>Build</u>

The build target executes the command eclipse.incrementalBuild which compiles all classes in the plug-in and places them into the directory specified in init.

<u>Export</u>

The export target copies necessary files to DPD's plug-in directory. The files copied correspond to the packages listed in the <requires> tag of the plugin.xml file. For example resources.jar from org.eclipse.core.resources_2.0.0 is copied to DPD's plug-in directory. The ant command **jar** is called on the directory that holds all DPD class files. In this case, the directory is **bin**. Lastly, this jar file is copied to DPD's plug-in directory.

Once the build file is run successfully, the user has to manually move DPD's plug-in directory into the eclipse directory. Eclipse must be shut down and restarted in order to register that the plug-in is there. In order to run DPD, one goes to File -> New -> Other -> DPD Wizard. A screen shot is shown in below


Java - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help
Package Explorer
MilosASE
RiverKayaking
C:\j2sdk1.4.0\jre\lib\rt.jar
ejbModule
source
TestDrivers
JRE_LIB - C:\Program Files\Java\j2re1.4.
C:\jboss-3.0.0_tomcat-4.0.3\client\jnp-c
C:\jboss-3.0.0_tomcat-4.0.3\client\jboss
C:\jboss-3.0.0_tomcat-4.0.3\client\jboss
C:\jboss-3.0.0_tomcat-4.0.3\client\jboss
C:\eclipse\plugins\org.eclipse.tomcat_4.0
C:\jboss-3.0.0_tomcat-4.0.3\client\jboss
C:\jboss-3.0.0_tomcat-4.0.3\client\log4j
C:\jboss-3.0.0_tomcat-4.0.3\client\jboss
C:\eclipse\plugins\org.junit_3.7.0\junit.ja
META-INF
Mail
lib
webApplication
build.xml
ShoppingNetwork
New
Select
DPD Wizards
DPD Wizards
Java
Plug-in Development
PSE-Application
Scheduled Tasks
Session Facade Wizard
Simple
ValueOjbectWizard
Design Pattern Developer Wizard
< Back
Next >
Finish
Cancel

Launching the Design Pattern Developer Wizard

Class Description

Rule Store Class Overview

As the Rule Store classes are consistent across all wizards, they are listed first, and not by wizard.

FileTypes.java – contains a list of all possible file types that might be used other than java files. This is typically for the purposes of implementing the client tier where the file types vary. At the time of writing the project includes only Java Server Pages.

JavaClassRules.java – establishes general formats for Java files such as ending in the string ".java".

JSPRules.java – establishes general formats for Java Server Pages such as a start or end flag (<% or %> respectively), an import line(<%@), some key words ("find" or "create") and an ending string to identify file names (.jsp)

Patterns.java – specifies a list of all patterns in the tool. This class is used in coordination with the wizard for user selection of a specific design pattern.

ProjectStructureRules.java – specifies the name of the project with which one is working as well as potential new folders to create should the user desire a new directory structure.

ServiceLocatorClass.java – specifies how to create a ServiceLocator class. The name of the class is specified as well as default methods that must be included. For example, in DPD, a method called connectToEntities must be created in the new Service Locator class which makes all lookups and connections to entity beans and does so for the entire J2EE application. A constructor for this class is created and the class is also made as a Singleton so as to avoid having to create a new instance each time the Service Locator tasks are required.

ServiceLocatorTemplate.java – this works with the ServiceLocatorClass.java to create a service locator file.

ServletRules.java – similar to JSPRules.java in that it specifies the requirements of a servlet file. For example, the naming convention is included here as being of the format SomeName_Servlet.java

SessionBeanBean.java – there are five files corresponding to the creation of a single session bean as the Session Façade. The SessionBeanBean.java file specifies the requirements of the actual bean class required to create any Session Bean. Required methods such as ejbActivate, Passivate, Create, Remove and get and set SessionContext are included in this file.

SessionBeanHomeInterface.java – the SessionBeanHomeInterface.java file specifies how to actually create the Home Interface required to implement any session bean. Here requirements such as "extends javax.ejb.EJBHome" are included as is the create method with the required exception.

SessionBeanRemoteInterface.java – specifies how to actually create the Remote interface required to implement any session bean. Requirements such as "extends javax.ejb.EJBObject" are specified.

SessionBeanTemplate.java – specifies the requirement of a bean implementation, a remote interface and a home interface for the creation of any session bean, and it creates these files. It also instantiates the final file required to create a session bean: the SessionBeanXMLFile.java

SessionBeanXMLFile.java – specifies what the deployment descriptor should entail for a session bean, in xml format.

SFUtilityClass.java – specifies the requirements for creating the actual Session Façade, and putting this creation into a single class.

Singleton.java – specifies the requirements for creating a singleton class.

ValueObjectRules.java – like JSPRules.java and ServletRules.java, this specifies the format of creating a value object. For example, it specifies a key id as being "VO" which is appended to the name of any file created as a value object.

Original Project Analysis Class Overview

What follows is a description of the classes in the Target Project Analysis package of each wizard. <u>Session Facade Value Object Wizard</u>

AnalyzeJSPFile.java – scans java server pages to find lines that contain a direct reference to the entity beans in a project. It removes direct references and replaces them with references to the session bean. It also removes any lookup information in the java server page.

AnalyzeServletFile.java – scans servlet files to find lines that contain a direct reference to the entity beans in a project. It removes direct references and replaces them with references to the session bean. It also removes any lookup information in the servlet page.

FindFileGroups.java – establishes the desired tier structure. Typically J2EE projects are grouped into three packages: one for the client tier (e.g. .jsp files), one for the integration tier (e.g. servlets) and one for the business tier (e.g. entity beans). If the original project does not follow a suitable structure (as per the developer's wishes) this file gives the option to reorganize the project, and also takes note of the .jsp, servlet and business files.

MappingDefinition.java – a generic class for handling the transfer of pertinent information from wizard page to wizard page.

Session Façade

The Target Project Analysis package for the Session Façade without the Value Object is very similar to that of the Session Façade with a value object pattern nested inside of it. The primary difference is in the return value of calls to the business tier. Rather than a finder method returning an entity bean or a vector of entity beans, when nesting the value object in the Session Façade pattern, a value object or a vector of value objects are returned, respectively. Thus the overall purpose of each class in the Target Project Analysis package is the same as above.

Value Object

AnalyzeJSPFile.java – scans java server pages to find lines that contain return values of entity beans or vectors of entity beans. It maintains the calls directly to the entity bean, but changes the return value to a value object instead of an entity bean.

AnalyzeServletFile.java – scans servlet pages to find lines that contain return values of entity beans or vectors of entity beans. It maintains the calls directly to the entity bean, but changes the return value to a value object instead of an entity bean.

FindFileGroups.java – same as described in Session Façade Value Object design pattern.

MappingDefinition.java – same as described in Session Facade Value Object design pattern.

Service Locator

AnalyzeJSPFile.java – scans java server pages to find service lookup information. References to the Java Naming Directory Interfaces 0 and related code are removed and replaced with a call to the Service Locator Class

AnalyzeServletFile.java – scans servlet pages to find service lookup information. References to the Java Naming Directory Interfaces 0 and related code are removed and replaced with a call to the Service Locator class.

FindFileGroups.java – same as described in Session Façade Value Object design pattern.

MappingDefinition.java – same as described in Session Façade Value Object design pattern.

Singleton

The singleton Target Project Analysis package only has the FindFileGroups.java class in the Target Project Analysis package and it functions in the same way as it does in the Session Façade Value Object pattern.

APPENDIX C

<u>Manual Re-factoring to Design Patterns</u>

<u>Questionnaire for Group A</u>

1. Record the start time from your computer:

2. Use the following chart to record your time for breaks and lunch.

	Start Time	Stop Time	Total Time
Break 1			
Lunch			
Break 2			

3. Examine the small J2EE application. You will see 6 entity beans, 9 servlets, and 12 jsp files.

4. Create one Session Façade class corresponding to the J2EE Session Façade pattern. Modify one entity bean to have a corresponding Value Object (you will have to create this value object).

5. Find the jsp pages that correspond to this entity bean, and make the required changes to access methods in the Session Façade instead of the entity bean, as per the Session Façade with nested Value Object pattern.

6. Find the jsp pages that correspond to this entity bean, and make the required changes to access methods in the Session Façade instead of the entity bean, as per the Session Façade with nested Value Object pattern.

7. Run the build.xml file to create the ear file. The target that should be run is localbuild_deploy. The file automatically places the ear file in the deployment directory.

8. Start up jboss. (tip: this will involve changing the service file, possibly named mysql-service.xml or RK-service.xml)

9. Run all test drivers using the file SNTest. Do all test drivers pass? Yes No

10. If No, how many tests failed?

11. Record the time that you start working on fixing the test drivers:
    a. Record the time that you have all test drivers passing and have completed the exercise:

<u>**Tool Re-factoring to Design Patterns**</u>

<u>**Questionnaire for Group B**</u>

12. Record the start time from your computer:

13. Use the following chart to record your time for breaks and lunch.

	Start Time	Stop Time	Total Time
Break 1			
Lunch			
Break 2			

14. Examine the small J2EE application. You will see 6 entity beans, 9 servlets, and 12 jsp files.

<u>The following will be performed by the tool.</u>

- Create one Session Façade class corresponding to the J2EE Session Façade pattern. Modify one entity bean to have a corresponding Value Object (you will have to create this value object).

- Find the jsp pages that correspond to this entity bean, and make the required changes to access methods in the Session Façade instead of the entity bean, as per the Session Façade with nested Value Object pattern.

- Find the jsp pages that correspond to this entity bean, and make the required changes to access methods in the Session Façade instead of the entity bean, as per the Session Façade with nested Value Object pattern.

15. Run the build.xml file to create the ear file. The target that should be run is localbuild_deploy. The file automatically places the ear file in the deployment directory.

16. Start up jboss. (tip: this will involve changing the service file, possibly named mysql-service.xml or RK-service.xml)

17. Run all test drivers using the file RKTest. Do all test drivers pass? Yes No

18. If No, how many tests failed?

19. Record the time that you start working on fixing the test drivers:
    a. Record the time that you have all test drivers passing and have completed the exercise:

Post-Experiment Questionnaire

1. What did you think of the tool used to refactor to design patterns? (Was it easy to use? Did it provide enough functionality to accomplish your task? Did it crash?) Please elaborate.

2. What did you think of the task to refactor to a design pattern manually? (Did you have enough time? Was the task too challenging/not challenging enough?) Please elaborate.

3. If you had a choice between using a tool like DPD and manually refactoring, which would you choose and why?

4. Would you like to see further implementation of the tool?

5. Please comment on anything else related to the experiment that we have missed.

6. How many years experience do you have with J2EE applications?

7. How many years experience do you have with design patterns?

8. Please record your reference number from the questionnaire sheet:

Post-Experiment Questionnaire - Answers

1. What did you think of the tool used to refactor to design patterns? (Was it easy to use? Did it provide enough functionality to accomplish your task? Did it crash?) Please elaborate.

It was easy to use since we had the tutorial and pretty much just selected defaults. I had difficulty deploying afterwards, it likely due to my inexperience.

2. What did you think of the task to refactor to a design pattern manually? (Did you have enough time? Was the task too challenging/not challenging enough?) Please elaborate.

This task was challenging for me. I did not finish. I've never used these patterns before and I have very limited J2EE programming experience. Since this was essentially my first time doing this, I would likely be able to do it much quicker.

3. If you had a choice between using a tool like DPD and manually refactoring, which would you choose and why?

I would choose to use the tool, since it would likely save time and typing.

4. Would you like to see further implementation of the tool?

Yes

5. Please comment on anything else related to the experiment that we have missed.

6. How many years experience do you have with J2EE applications?

1 part time month

7. How many years experience do you have with design patterns?

1 part time year

8. Please record your reference number from the questionnaire sheet: *1*

<u>Post-Experiment Questionnaire - Answers</u>

1. What did you think of the tool used to refactor to design patterns? (Was it easy to use? Did it provide enough functionality to accomplish your task? Did it crash?) Please elaborate.

Worked as expected. Did not crash. Functionality was adequate. Not enough feedback while it was working and when it was done. Concerned that it is sensitive to the code at the beginning, i.e. What if no servlets or .jsps were used?

2. What did you think of the task to refactor to a design pattern manually? (Did you have enough time? Was the task too challenging/not challenging enough?) Please elaborate.

Didn't have enough time. As expected challenge-wise (I knew it would be tough). Looking at the problem I would not have estimated 1 hour, probably 2.5 hours including more tests.

3. If you had a choice between using a tool like DPD and manually refactoring, which would you choose and why?

DPD, provided the starting point was appropriate.

4. Would you like to see further implementation of the tool?

Yes, Generating from scratch, too?

5. Please comment on anything else related to the experiment that we have missed.

The naming convention used in the starting [application] helped during the wizard run. If I was unfamiliar with code and it had no naming convention, it would be hard to use. Given one file, it could find the others? Also, the choice to eliminate the entity beans and use POJO data access objects instead might be useful for some.

6. How many years experience do you have with J2EE applications? *3*
7. How many years experience do you have with design patterns? *4+*
8. Please record your reference number from the questionnaire sheet: *3*

Post-Experiment Questionnaire - Answers

1. What did you think of the tool used to refactor to design patterns? (Was it easy to use? Did it provide enough functionality to accomplish your task? Did it crash?) Please elaborate.

A bit flaky but succeeded on the 3rd attempt. First 2 times it didn't crash, just quietly did nothing. (Maybe I missed a setting).

2. What did you think of the task to refactor to a design pattern manually? (Did you have enough time? Was the task too challenging/not challenging enough?) Please elaborate.

Never used Eclipse or jboss before. Actually getting the tests to run initially took most of the time, then [insufficient] left to complete it. This type of task is dependant on familiarity with the machine, tools, source structure etc., this dominated the time.

3. If you had a choice between using a tool like DPD and manually refactoring, which would you choose and why?

Manual. I come from the "code Crafter" school rather than the "test and refactor" school. So, chances are, I wouldn't need it on good code, but on bad code I'd worry about safety. (If the code is bad, how can you trust the tests?)

4. Would you like to see further implementation of the tool? *No.*

5. Please comment on anything else related to the experiment that we have missed.

If the two projects worked out of box (i.e. no configuration problems) that might isolate the tool/non-tool differences better.

6. How many years experience do you have with J2EE applications? *4*
7. How many years experience do you have with design patterns?

N/A (Know some of them but don't use them explicitly)

8. Please record your reference number from the questionnaire sheet: *4*

Post-Experiment Questionnaire - Answers

1. What did you think of the tool used to refactor to design patterns? (Was it easy to use? Did it provide enough functionality to accomplish your task? Did it crash?) Please elaborate.

The tool seemed fine – just deployment issues. Maybe it should add the lines it need automatically to the deployment descriptor files.

2. What did you think of the task to refactor to a design pattern manually? (Did you have enough time? Was the task too challenging/not challenging enough?) Please elaborate.

Not too difficult for one bean with only a few files. For very disorganized code, it would be very time consuming, especially with lots of beans and lots of fields.

3. If you had a choice between using a tool like DPD and manually refactoring, which would you choose and why?

Would depend on [the] situation. Probably use the tool and then manually adjust afterwards.

4. Would you like to see further implementation of the tool?

Yes. Things like this are good for such complex tasks.

5. Please comment on anything else related to the experiment that we have missed.

As always with J2EE deployment was biggest problem. I remembered why I don't do this stuff anymore ☺

6. How many years experience do you have with J2EE applications? *2*

7. How many years experience do you have with design patterns? *0*

8. Please record your reference number from the questionnaire sheet: *5*

Post-Experiment Questionnaire - Answers

1. What did you think of the tool used to refactor to design patterns? (Was it easy to use? Did it provide enough functionality to accomplish your task? Did it crash?) Please elaborate.

Yes it was easy to use. Sometimes forgot to press Next but pressed Finish instead.

2. What did you think of the task to refactor to a design pattern manually? (Did you have enough time? Was the task too challenging/not challenging enough?) Please elaborate.

I don't know beans well so if I had started with A first [manually refactoring] it would have been a lot more difficult.

3. If you had a choice between using a tool like DPD and manually refactoring, which would you choose and why?

DPD because there would most likely be less error if it is automated.

4. Would you like to see further implementation of the tool?

Yes

5. Please comment on anything else related to the experiment that we have missed.

I don't think my skill level was appropriate for this experiment (both in terms of J2EE & design patterns)

6. How many years experience do you have with J2EE applications? *< 1*

7. How many years experience do you have with design patterns? *1*

8. Please record your reference number from the questionnaire sheet: *6*

<u>Post-Experiment Questionnaire - Answers</u>

1. What did you think of the tool used to refactor to design patterns? (Was it easy to use? Did it provide enough functionality to accomplish your task? Did it crash?) Please elaborate.

It is easy to use and didn't crash. It should have also generated JUnit tests to assure the refactored code is working as before; update deployment files of the new Session Façade.

2. What did you think of the task to refactor to a design pattern manually? (Did you have enough time? Was the task too challenging/not challenging enough?) Please elaborate.

It is very tedious even [though] the task is very simple. It is also very time consuming

3. If you had a choice between using a tool like DPD and manually refactoring, which would you choose and why?

A tool as long as it also generates JUnit stub classes which fails by default this will force me to write the test drivers to assure quality of the refactored code. I must say I would not rely totally on any tools for complex refactoring. There [is] still need for human intervention

4. Would you like to see further implementation of the tool?

5. Please comment on anything else related to the experiment that we have missed.

Lots of time being spent to [configure]e the environment.

6. How many years experience do you have with J2EE applications? *1.5 yr.*

7. How many years experience do you have with design patterns? *½ - 1 yr.*

8. Please record your reference number from the questionnaire sheet: *7*

Post-Experiment Questionnaire - Answers

1. What did you think of the tool used to refactor to design patterns? (Was it easy to use? Did it provide enough functionality to accomplish your task? Did it crash?) Please elaborate.

Changing Façade Name didn't work in SFHome. JSPs were not totally refactored. Deploy descriptors also not yet automatically updated

2. What did you think of the task to refactor to a design pattern manually? (Did you have enough time? Was the task too challenging/not challenging enough?) Please elaborate.

It was OK. I have some experience at this.

3. If you had a choice between using a tool like DPD and manually refactoring, which would you choose and why?

Manually

- tool is not advanced enough yet…
- no surprises, opportunity to review code.

4. Would you like to see further implementation of the tool?

Sure.

5. Please comment on anything else related to the experiment that we have missed.

ANT files may have bugs?

Pizza in same room = torture.

6. How many years experience do you have with J2EE applications? *3*

7. How many years experience do you have with design patterns? *3*

8. Please record your reference number from the questionnaire sheet: *8*

<u>Post-Experiment Questionnaire - Answers</u>

1. What did you think of the tool used to refactor to design patterns? (Was it easy to use? Did it provide enough functionality to accomplish your task? Did it crash?) Please elaborate.

It worked almost flawlessly for me. It did what it was supposed to . Only problem was with how it refactored one of the jsp's improperly.

2. What did you think of the task to refactor to a design pattern manually? (Did you have enough time? Was the task too challenging/not challenging enough?) Please elaborate.

Almost finished, was getting some strange ClassCastException. Was much more difficult than using the tool, took a lot more time too.

3. If you had a choice between using a tool like DPD and manually refactoring, which would you choose and why?

Tool, much easier, saves time and energy.

4. Would you like to see further implementation of the tool?

More patterns would be nice.

5. Please comment on anything else related to the experiment that we have missed.

6. How many years experience do you have with J2EE applications? 2.5

7. How many years experience do you have with design patterns? 1

8. Please record your reference number from the questionnaire sheet: 9