Dr. Jones: A Software Design Explorer's Crystal Ball

by

Mark A. Foltz

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 2003

© Massachusetts Institute of Technology 2003. All rights reserved.

Author . . . .
Department of Electrical Engineering and Computer Science
August 22, 2003

Certified by . . . .
Randall Davis
Professor of Computer Science, MIT
Thesis Supervisor

Certified by . . . .
Daniel Jackson
Associate Professor of Computer Science, MIT
Thesis Reader

Certified by . . . .
Howard Shrobe
Principal Research Scientist, MIT
Thesis Reader

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Dr. Jones: A Software Design Explorer's Crystal Ball

by

Mark A. Foltz

Submitted to the Department of Electrical Engineering and Computer Science
on August 22, 2003, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering

Abstract

Most of software design is redesign. Redesign in the normal course of design happens when the software becomes difficult to maintain and the problem it is intended to solve has changed. Although software redesign is necessary, frequent, and pervasive, there is a dearth of tools that help programmers do it. Instead, programmers primarily use pen and paper, away from the computer where tools could help the most. To address this shortcoming, I have developed DR. JONES, a redesign assistant for Java programs.

DR. JONES diagrams the class structure of a Java program and allows the programmer to modify that design by applying refactorings. Refactorings are localized patterns of structural change intended to improve a program's design, without changing its observable behavior. With DR. JONES, the programmer can explore the design space of the program, inspect future designs as visual diagrams, and get design assistance to guide his refactoring choices.

As the programmer explores designs, DR. JONES explicitly maps the design space he traverses. This map lets him revisit any prior design and branch to explore an alternative design path, without having to explicitly manage versions of the program.

DR. JONES is distinguished from other refactoring tools by separating the tasks of developing an improved design through design exploration from transforming the source code to execute design changes. It does so by deriving and using an abstract representation of the program that captures the essential information needed for design exploration, while omitting its source-level details. DR. JONES also characterizes refactorings in a novel manner suitable for interactive design exploration. Twenty-two such refactorings are incorporated into the DR. JONES prototype.

This research also contributes user interface techniques for software design exploration, including multiple-level-of detail rendering for software design diagrams, and a dialogue management interface for DR. JONES' design assistance.

Thesis Supervisor: Randall Davis
Title: Professor of Computer Science, MIT

Thesis Reader: Daniel Jackson
Title: Associate Professor of Computer Science, MIT

Thesis Reader: Howard Shrobe
Title: Principal Research Scientist, MIT

Acknowledgments

I would like to acknowledge some of the many persons and groups who supported me during this undertaking. Without them, this thesis would never have been possible.

First, thanks to my mother Susan, my father Alan, my brother Josh, and my grandmother Mildred for their love, their kind words of encouragement and advice, and their many shipments of homemade cookies.

Randy Davis has been a gifted advisor and mentor. His encouragement, approachability, and patience have served as my model of how an advisor should interact with students. His insights and direction have, more than anything else, shaped my approach to research problems.

Howie Shrobe greatly enlightened me about the AI Lab's rich traditions in LISP programming environments as well as the Programmer's Apprentice project. His feedback helped me place this research in the larger context of what it means to redesign software.

Daniel Jackson's keen intellect was instrumental in sharpening my focus and clarifying my ideas. As a result of his input, those ideas and their exposition improved immeasurably.

Allison Waingold in the LCS Software Design Group unselfishly shared her Womble object model extractor with me and wrote the code necessary to integrate it with DR. JONES.

My fellow students in the Design Rationale Group served as an ever-available sounding board. They are all great listeners and a source of much-needed comic relief.

Jonathan Bachrach of the Dynamic Languages Group at the AI Lab shared his knowledgeable perspectives on programming languages and software engineering.

Michael Ernst and Peggy Storey (University of Victoria, British Columbia) lent an ear to some of my early ideas for DR. JONES and pointed me to some invaluable resources.

Thanks to Rod Brooks and everyone in the Artificial Intelligence Lab for creating an unequaled intellectual playground. It's been a privilege to study and learn here.

Finally, thanks to my sponsors, including those in the Project Oxygen alliance.

Contents

1 Introduction **11**
1.1 Thesis . . . 11
1.2 A Crystal Ball for Software Design Exploration . . . 12
1.3 Summary of Contributions . . . 13
1.4 Scenario: Building Bicycles . . . 14
1.5 Roadmap . . . 21

2 The Role of a Redesign Assistant **23**
2.1 What Is The "Design" of a Program . . . 24
2.2 Why Is Software Redesigned? . . . 25
2.3 Software Redesign Tasks . . . 26
2.3.1 Understanding the Program . . . 27
2.3.2 Diagnosing Design Flaws . . . 27
2.3.3 Planning Redesign . . . 28
2.3.4 Executing the Plan . . . 28
2.4 Dividing The Labor: Building Bicycles Revisited . . . 29
2.4.1 Pen-and-Paper and a Text Editor . . . 29
2.4.2 Source Code Refactories . . . 31
2.4.3 Dr. Jones . . . 32
2.5 Freedom Versus Power . . . 33
2.6 Summary . . . 35

3 Redesigning With Refactorings **37**
3.1 Refactorings: An Overview . . . 37
3.1.1 Refactorings Are Local, Structural Changes . . . 37
3.1.2 Refactorings Are Dr. Jones' Redesign Language . . . 38
3.2 Refactoring Knowledge . . . 39
3.2.1 The Knowledge in the Refactoring Browser . . . 40
3.2.2 The Knowledge in Dr. Jones . . . 41
3.3 Organizing The Space of Refactorings . . . 42
3.4 Summary . . . 45

4 The Refactoring Knowledge Base 47
4.1 What Refactorings Dr. Jones Knows 47
4.2 What Dr. Jones Knows About a Refactoring 48
4.2.1 Must-Guards 51
4.2.2 Should-Guards 52
4.2.3 Design Instance Transformation 53
4.2.4 Design Suggestions 53
4.2.5 Source Editing Instructions 53
4.3 Knowledge Engineering of Dr. Jones 54
4.3.1 Knowledge Base Development 54
4.3.2 Adding New Refactorings 55
4.4 Composing Refactorings 57
4.5 Summary 58

5 Dr. Jones' Design Representation 59
5.1 The Design Instance Representation 59
5.1.1 A Design Instance Is A Graph 60
5.1.2 Dependency Types 62
5.2 Building the Initial Design Instance 62
5.2.1 JavaDoc Analysis 62
5.2.2 Bytecode Analysis 63
5.2.3 SuperWomble Analysis 63
5.2.4 Departures From The Java Specification 63
5.3 The Design Space Representation 64
5.4 Comparing Program Representations 64
5.4.1 Abstract Syntax Trees 65
5.4.2 FAMIX 65
5.4.3 EMF 66
5.4.4 Reflexion Models 66
5.4.5 The Plan Calculus and CAKE 66
5.5 Capturing Design Rationale 67
5.6 Summary 68

6 Exploring Designs With Dr. Jones 69
6.1 Visualizing The Current Design 70
6.1.1 Multiple Level-of-Detail Rendering 71
6.1.2 Determining an Element's Level of Detail 73
6.2 Interpreting the Programmer's Input 74
6.3 Managing Dialogue 75
6.3.1 The Dialogue Tree 75

6.4 Navigating The Design Space . . . 77
6.4.1 The Design Space Map . . . 77
6.4.2 Bookmarks and the To Do List . . . 77
6.5 Additional User Interface Capabilities . . . 79
6.5.1 Inspecting The Design Space . . . 79
6.5.2 Source Inspection . . . 79
6.6 Current Limitations of the User Interface . . . 80
6.7 Summary . . . 81

7 Two Design Exploration Scenarios 82
7.1 Decomposing Bicycle . . . 82
7.1.1 Decomposition By Subclassing . . . 83
7.1.2 Decomposition By Delegation . . . 85
7.1.3 Lessons Learned . . . 88
7.2 Evolving JUnit . . . 90
7.2.1 Generalizing the Simple Framework . . . 90
7.2.2 Adding Test Variants . . . 91
7.2.3 Lessons Learned . . . 94
7.3 Summary . . . 96

8 Conclusion 97
8.1 Thesis . . . 97
8.2 Summary of Contributions . . . 98
8.3 Related Work . . . 99
8.3.1 Software Evolution Environments . . . 99
8.3.2 Program Understanding and Visualization . . . 100
8.3.3 Refactoring Theory . . . 100
8.3.4 Refactoring Development Environments . . . 101
8.3.5 Extreme Programming . . . 101
8.4 Future Work . . . 102
8.4.1 Addressing Lessons Learned . . . 102
8.4.2 Broadening Dr. Jones' Refactoring Abilities . . . 103
8.5 Software Redesign, Broadly Considered . . . 105
8.5.1 Natural Interaction for Redesign . . . 105
8.5.2 Redesigning Behaviors and Interfaces . . . 105
8.5.3 A New Programming Language . . . 106

A The Refactoring Space 107

B The Refactoring Knowledge Base 110

C A Refactoring Implementation **134**

D The Bicycle Redesign Scenario **139**
D.1 Source Code 139
D.2 Refactoring Script 142

E The JUnit Design Scenario **145**
E.1 Source Code 145
E.2 Refactoring Script 146

List of Figures

1-1 A sketch of a software system's design made by a programmer maintaining it. 12
1-2 The initial design of the Bicycle class. 15
1-3 The Bicycle design after Step 2. 17
1-4 The Bicycle design after Step 3. 17
1-5 The Custom... subclasses share a common API and thus can be generalized. 18
1-6 The first design alternative for the Bicycle class. 19
1-7 The second design alternative for the Bicycle class. 19
1-8 The map of the design space explored by Ecks. 20

2-1 A model of the tasks in software redesign. 27
2-2 A hand-drawn diagram of the Bicycle class. 29
2-3 Design alternatives that Ecks would draw with pen and paper 30
2-4 With pen-and-paper redesign, the computer offers limited support for design evolution. 31
2-5 Source code refactories benefit the programmer after design decisions have been made. 32
2-6 DR. JONES lets the programmer rapidly complete design iterations without manipulating the source code. 34
2-7 A qualitative comparison of the freedom versus power afforded by toolsets for redesign. 35

4-1 The entries in the refactoring knowledge base. 48
4-2 The graph of dependencies in the knowledge base. 49
4-3 The knowledge base entry for the Specialize aTypeCodeField refactoring. . . . 50
4-4 Part of a prototype entry for Rename Method in the Dr. Jones knowledge base. 55
4-5 The class template for refactorings. 56

5-1 The containment and dependency relationships among nodes. 61
5-2 A fragment of the design space for the Bicycle example. 65
5-3 The design space representation extended to support design deliberation. . 67
5-4 A deliberation that might have happened regarding the two designs in the Bicycle scenario. 68

6-1 The initial rendering of the Bicycle class in Dr. Jones. 70
6-2 An overly complex design diagram produced by a commercial development tool. 71
6-3 The minimum (left), default (middle), and maximum (right) levels of detail for a class. 72
6-4 The command console for Dr. Jones. 74
6-5 The console after the dialogue for the Specialize refactoring is complete. . . . 75
6-6 The dialogue tree generated for the complete Specialize dialogue. 76
6-7 The Design Space Map lets the programmer see and navigate the design space explored during a redesign session. 78
6-8 Bookmarking allows the programmer to create named designs. 79
6-9 The Design Space Browser 80

7-1 The initial design of the Bicycle class. 84
7-2 Bicycle decomposed into subclasses by frame type. 85
7-3 The subclassing redesign alternative for Bicycle. 86
7-4 frameType encapsulated into its own class hierarchy. 87
7-5 The delegation design alternative for Bicycle. 87
7-6 The kernel framework for JUnit. 90
7-7 The evolved design of the JUnit framework. 92
7-8 A JUnit design alternative with test variants. 93
7-9 The design space of the JUnit framework. 94
7-10 MoneyTest implemented in the JUnit framework. 95
7-11 Dr. Jones prevents us from decomposing run() in MoneyTest. 96

8-1 Pulling up methods in Eclipse requires detailed, source-level interaction. . . 102

List of Tables

3.1 A set of refactoring verbs that correspond to generic refactoring actions. . . 43
3.2 The refactoring space, with points checked that are meaningful refactorings. 44
3.3 The overlap between the refactoring space and *Refactoring*. 44

5.1 Node properties in the design instance representation. 61
5.2 The dependency types in the design instance representation. 62

6.1 The levels of detail available for rendering a class in DR. JONES. 73

A.1 The Refactoring Space: A Comparison 108

Chapter 1

Introduction

1.1 Thesis

Program design exploration should – and can – be supported by the computer. Most of software design is redesign, and although programmers frequently redesign software, there is a dearth of tools that help them do it. The tools that do exist primarily automate the transformation of source code, instead of helping the programmer make higher-level design decisions. What is needed is a tool that helps the programmer see and explore the design space of the program. Without one, program redesign will stay on pen and paper, where the computer cannot help.

The key to accomplishing this is to separate the task of helping a programmer plan a program's redesign from the task of carrying out redesign steps (i.e., transforming the source code). To demonstrate this idea, I have created DR. JONES, a design exploration assistant. DR. JONES lets the programmer refactor the design of Java programs. It does so by first building an abstract representation of the program suitable for redesign. The programmer can then apply DR. JONES' refactorings to see potential future designs (making it the design explorer's "crystal ball").

Unlike tools which only transform source code, DR. JONES lets the programmer explore multiple design alternatives, see the results of refactorings in concise UML-like diagrams, and obtain design guidance while refactoring. DR. JONES fundamentally separates the task of planning redesign from the task of executing redesign, and collaborates with the programmer as he redesigns.

1.2 A Crystal Ball for Software Design Exploration

It is a mistake to think of a piece of software as a static entity. The source code of a program is simply a snapshot of an artifact that is always evolving. Some simple programs, or programs at the end of their life cycle, rarely need to be modified. But nearly all useful programs are under constant pressure to change. This pressure leads to the continuous and pervasive modification of existing software, an activity which consumes a large portion of programmers' time (Griswold and Notkin, 1993).

Despite the pervasive nature of software change, there are few tools to help programmers do it. This is particularly true for changes to the high-level design of the software, such as the basic abstractions that organize the program.

When programmers want to redesign software at that level, they typically use a cumbersome manual process. The first step – getting a clear picture of the current design – typically requires the programmer to manually reverse engineer the program into hand-drawn diagrams (such as Figure 1-1). Next, problems and redesign moves are noted on these diagrams. Finally, the programmer returns to the source and implements his redesign using his diagrams as a guide. If further redesign is needed – or the redesign plan has to be rethought mid-stream – this process must begin again.

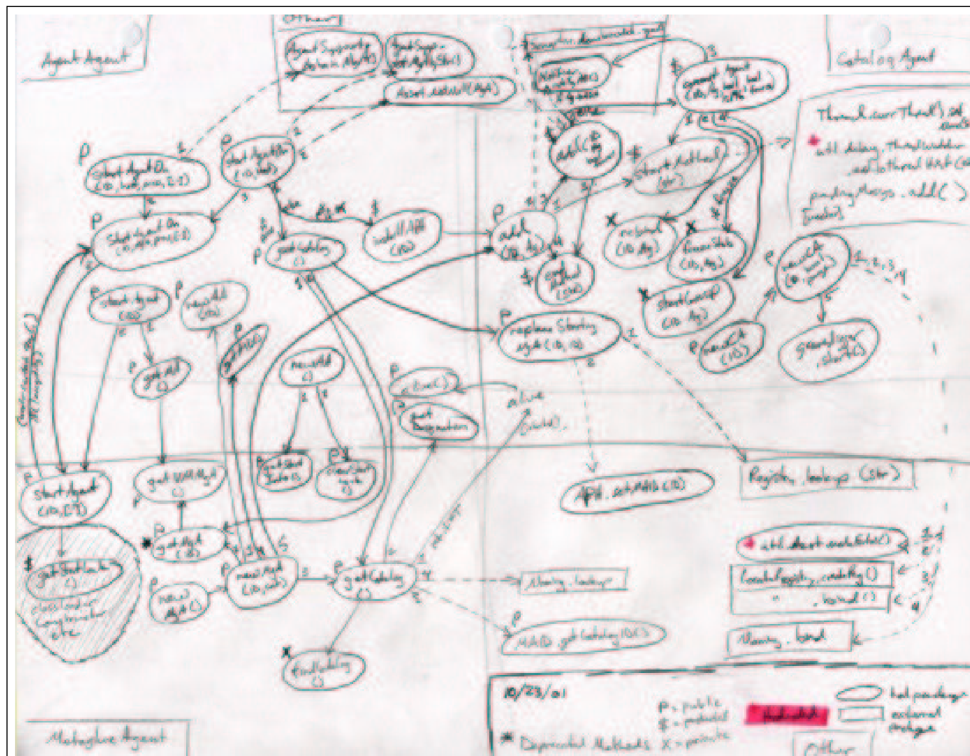

Figure 1-1: A sketch of a software system's design made by a programmer maintaining it.

I bring the computer back into this process by creating a "crystal ball" for the programmer. This crystal ball lets the programmer do some of the design exploration usually

done with pen and paper, with the added benefit of the computer's assistance. It lets the programmer take a step back and see the organization of his program at a high level. He then can explore the design space of the program by proposing design changes to the computer, to which the computer responds by showing him the resulting future designs. The computer also watches "over the programmer's shoulder" and makes suggestions of additional design moves to improve the working design. Throughout this process, all of the alternative designs considered are recorded and the programmer can easily revisit ones he finds most promising. Once he chooses a (presumably) better design, the computer could then help him transform the source to implement it (although that process is not the focus of this work).

This thesis will describe DR. JONES,[1] a software redesign tool intended to be that crystal ball. DR. JONES lets the programmer explore the design space of Java programs using a set of twenty-two refactorings. (A refactoring is a localized pattern of structural change that improves a program, without changing its visible behavior (Fowler, 1999).) DR. JONES lets the programmer see diagrams of current and potential future designs, explore multiple design alternatives, and obtain feedback and guidance throughout the redesign process.

1.3 Summary of Contributions

This thesis makes several contributions to research in software redesign tools, incorporating them into a prototype that illustrates their use. Those contributions include:

- *Separating the concern of design exploration from source code transformation.* They are really two separate tasks. The goal of design exploration is to find an improved design for the program. The goal of source code transformation is to realize a new design while not introducing new errors into the program. In an analogy with the domain of architecture, DR. JONES is a programmer's drafting table, paper, and pencils; source code transformers are his saw and hammer.

  In reality, programmers often cycle between these two levels of maintenance, but current tools do not make a clear distinction between them. This thesis argues the programmer benefits when that distinction is made.

- *An abstract program representation suitable for redesign.* Because DR. JONES is not concerned with source code transformation, it works with an abstract description of the program that hides many of its source-level details. This thesis contributes such a design-level representation for Java programs. This design-level representation captures enough information about the program for DR. JONES to give the programmer useful guidance during redesign. It also records the design space the programmer has explored – which design alternatives were considered, and how they are related.

[1] Named after the famous, fictional explorer.

- *A knowledge base of refactorings written specifically for interactive redesign.* Prior work on refactoring has contributed descriptions of them at varying levels of formality and detail (Fowler, 1999; Opdyke, 1992), with the emphasis on the preconditions that ensure their validity and the source code modifications that implement them. This thesis contributes a knowledge base of twenty-two refactorings written specifically for the purpose of interactive redesign. This knowledge base includes additional kinds of knowledge not found in these other descriptions, such as refactorings that may be required or suggested by a proposed refactoring.

- *A user interface for software design exploration.* DR. JONES provides a novel user interface for the interactive redesign of Java programs. This thesis contributes interface components to render design diagrams at multiple levels of detail, manage the redesign dialogue between the programmer and DR. JONES, and assist the programmer in navigating the design space he has explored.

1.4 Scenario: Building Bicycles

A scenario will help to illustrate the kind of design exploration supported by DR. JONES. Here, I present it at a high level of detail. Later more details will be presented to illustrate other aspects of DR. JONES.

Suppose that Ecks Presso, our hero, is working on a program for Kannondell, a bicycle manufacturer. The manufacturer makes bicycles built on three different kinds of frames: mountain frames, for rugged off-road use; road frames, for smooth paved roads; and hybrid frames, which combine aspects of mountain and road frames. Bicycles may be built in one of several pre-configured models, or a customer may order a custom configuration by picking and choosing a frame and components.

The program currently uses a single Bicycle class to represent a bicycle configuration, which DR. JONES renders in Figure 1-2. This class has several responsibilities: storing information about the frame and components making up the configuration; providing a way to customize the configuration; and checking the compatibility of the bicycle's frame and components. A large class like this one with many methods and instance variables is hard to maintain and extend. It is a classic antipattern (Beck and Fowler, 1999), or design flaw, and it begs to be decomposed. Ecks investigates this class in more detail and finds these specific problems:

1. The legal combinations of components depend primarily on the frame type. Although we don't show the source for checkSpecs, Ecks finds it consists of a top-level switch statement that has a different case according to the value of the frameType field, which can be MOUNTAIN, HYBRID, or ROAD.

2. The getSuspension and setSuspension methods make sense for hybrid and mountain bikes, but not road bikes, which cannot have a suspension.

3. The methods which set bicycle properties (like setWheelSize) should only be available for custom bicycle configurations, not pre-built configurations.

edu.mit.infoarch.drjones.bicycle

Bicycle

A class representing a bicycle at the Kannondell website

checkSpecs():boolean
Return true iff this bicycle configuration is valid.

getFrameType():int
Return the frame type of this bicycle.

getNumGears():int
Return the total number of gears in this bike.

getPrice():float
Return the price of this configuration.

getSuspension():int
Return the suspension value

getWheelSize():int
Return the wheel size (in cm).

isCustom():boolean
Returns true iff this is a custom bike.

setCustom(boolean):void
Set the custom field.

setFrontGears(int):void
Sets the value of frontGears

setRearGears(int):void
Sets the value of rearGears

setSuspension(int):void
Set the value of suspension

setWheelSize(int):void
Set the wheel size (in cm).

HYBRID:int
Constant for hybrid bike frame type.

MOUNTAIN:int
Constant for mountain bike frame type.

ROAD:int
Constant for road bike frame type.

frameType:int
The frame type for this bike.

frontGears:int
The number of front gears.

isCustom:boolean
True if this bike is a custom bike (built by user).

rearGears:int
The number of rear gears.

suspensionType:int
The suspension type for this bicycle.

wheelSize:int
The wheel size in cm.

Figure 1-2: The initial design of the Bicycle class.

After noting these problems Ecks starts redesigning the class with DR. JONES by loading its source and compiled forms into the tool. He solves the four problems in the list as follows:

1. To attack the first problem in the list, he needs a way to separate the behaviors in Bicycle that depend on the frame type. One way to do so is to subclass Bicycle. Ecks takes this approach, and tells DR. JONES to Specialize the Bicycle class according to the values of the frameType field. DR. JONES cannot tell automatically which values frameType can take, so Ecks also specifies the MOUNTAIN, HYBRID, and ROAD constant fields. DR. JONES responds by creating Mountain, Hybrid, and Road subclasses for Bicycle. It also removes the frameType field, and the MOUNTAIN, HYBRID, and ROAD constant fields, because they aren't needed any more.

   DR. JONES then makes a design suggestion: Push Down checkSpecs(), since that method made use of the frameType field, and so its behavior likely depended on its value. Ecks tells DR. JONES to follow through on this suggestion, but it turns out the
   Push Down refactoring can't be done right away – checkSpecs() is a private method, and it is not possible to push down a private method in Java.

   Fortunately, DR. JONES replies with a way to make the refactoring possible: first Expose checkSpecs(), widening its visibility from private to default. Ecks agrees to this fix, and the original goal of Push-ing Down checkSpecs() can be accomplished. The design resulting from this first interaction is shown in Figure 1-3[2]

2. Ecks now moves on to the second problem in the list. He wants to make suspension-related methods only available to bicycles that can have suspension. Now that Bicycle has subclasses for each frame type, this becomes possible. He tells DR. JONES to Push Down the getSuspension and setSuspension methods into its Mountain and Hybrid subclasses. DR. JONES renders the end result in Figure 1-4.

3. Ecks now wishes to address the third problem. To do so, he needs to create further subclasses for customizable bicycles, and use them to isolate the methods that set bicycle parameters (setFrontGears(), setWheelSize(), etc.). Because bicycles with any frame can be customized, he tells DR. JONES to Specialize Mountain *to* CustomMountain, Specialize Hybrid *to* CustomHybrid, and Specialize Road *to* CustomRoad. He can then Pull Down the setter methods into these new subclasses, arriving at the design in Figure 1-5.

4. Ecks, looking at Figure 1-5, notes that the new Custom... subclasses share a common group of setter methods for bicycle properties. (This kind of insight would have been difficult to see in the initial design, or by doing pen and paper redesign.) He tells DR. JONES to Generalize CustomMountain, CustomHybrid, and CustomRoad into a

[2] In these diagrams, unlabeled arrows indicate inheritance (is-a) relationships between classes, and labeled arrows indicate multiplicity (has-a) relationships. A "!," "?," or "*" appended to a label means the multiplicity is exactly one, zero-to-one, or zero-to-many, respectively. The presence or absence of dashes on the arrows is not significant.

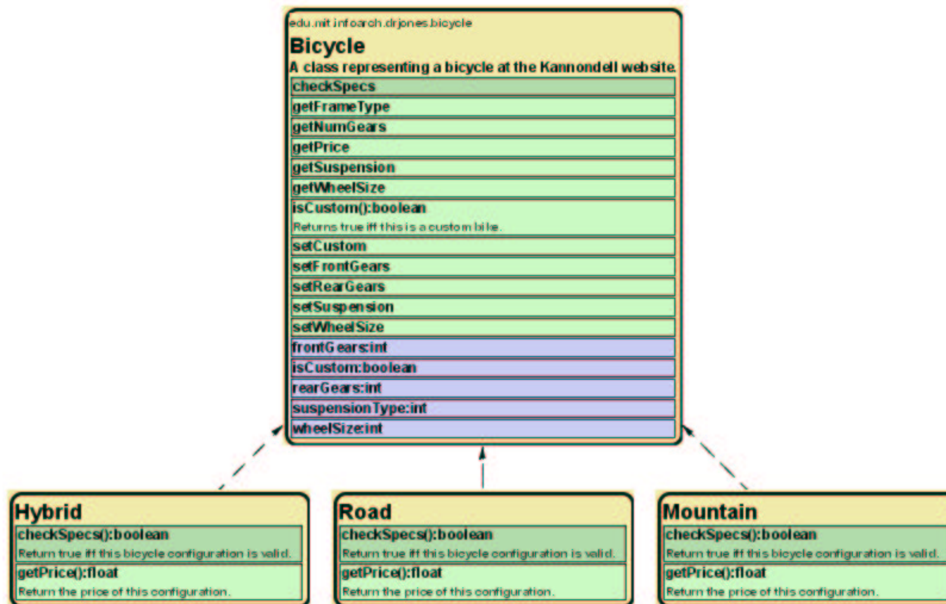

Figure 1-3: The Bicycle design after Step 2.

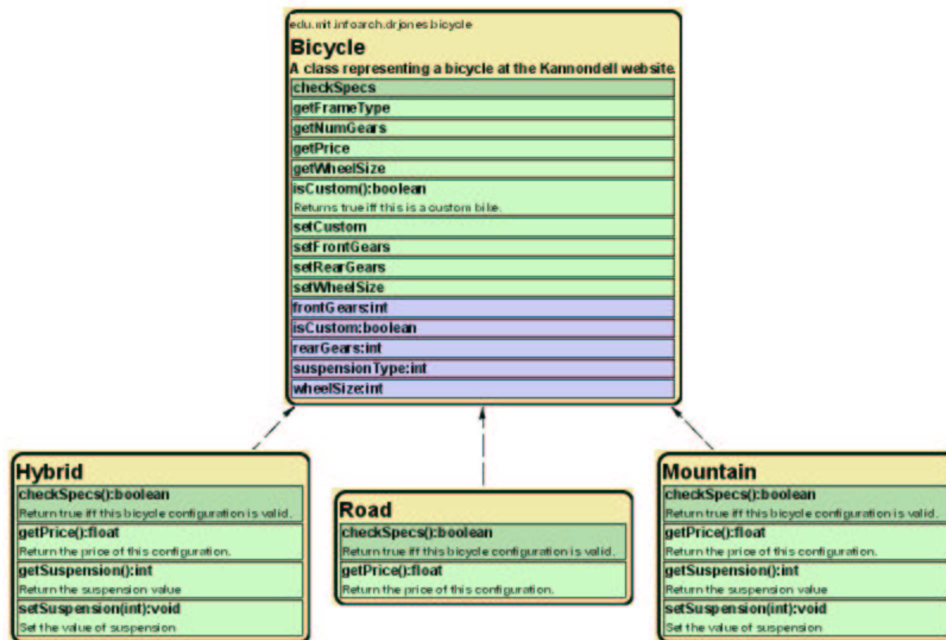

Figure 1-4: The Bicycle design after Step 3.

Customizable interface. DR. JONES does so, and then suggests that Ecks Pull Up the methods the three classes have in common. Ecks follows DR. JONES' suggestions and finally arrives at the design in Figure 1-6.

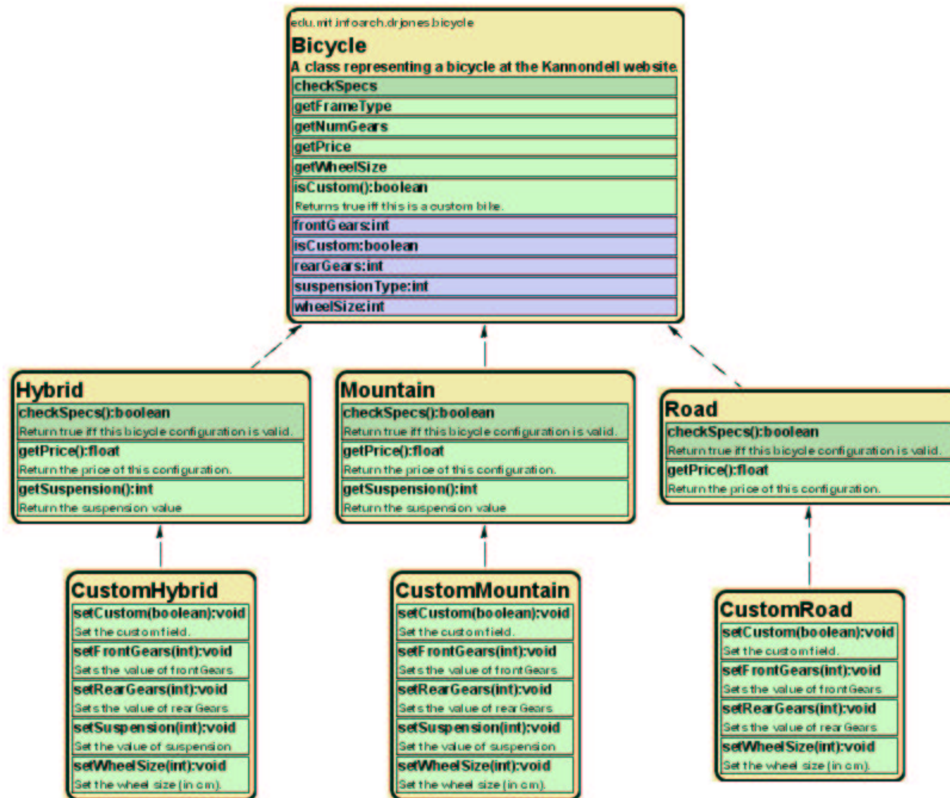

Figure 1-5: The Custom... subclasses share a common API and thus can be generalized.

This proposed redesign addresses the problems Ecks found with the initial design, but it is not the only way to proceed. Because DR. JONES has kept a record of the design space – all the design alternatives considered – Ecks can easily return to the initial design and try a new tack.

His first alternative assumed that the best way to represent behavior specific to the frame type was to create a subclass of Bicycle for each frame type. Instead, Ecks wonders, what if a Frame were considered a component of a Bicycle, and in turn Frame had Mountain, Hybrid, and Road subclasses?

Omitting the specific steps involved, this new assumption leads Ecks to the equally plausible alternative shown in Figure 1-7. Both of these designs exhibit better modularity, a clearer division of responsibilities, and more opportunities for reuse when new features are demanded of the program.

DR. JONES keeps track of both these alternatives as well as the intermediate steps Ecks took to reach them. The result of the session is a map of the design space Ecks has explored (Figure 1-8), which he can use to revisit any previous design and further explore the design

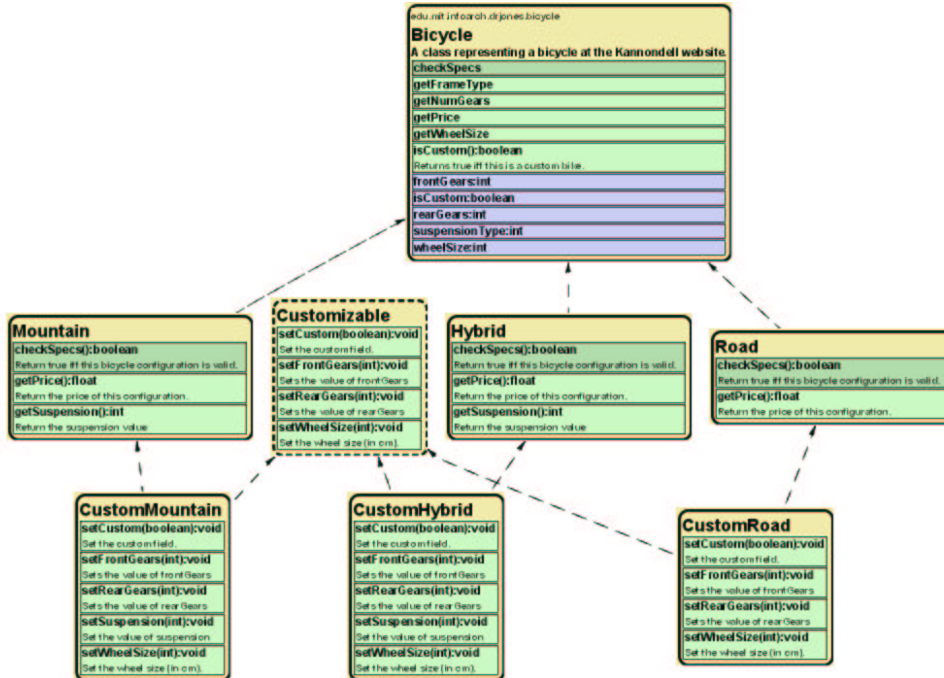

Figure 1-6: The first design alternative for the Bicycle class.

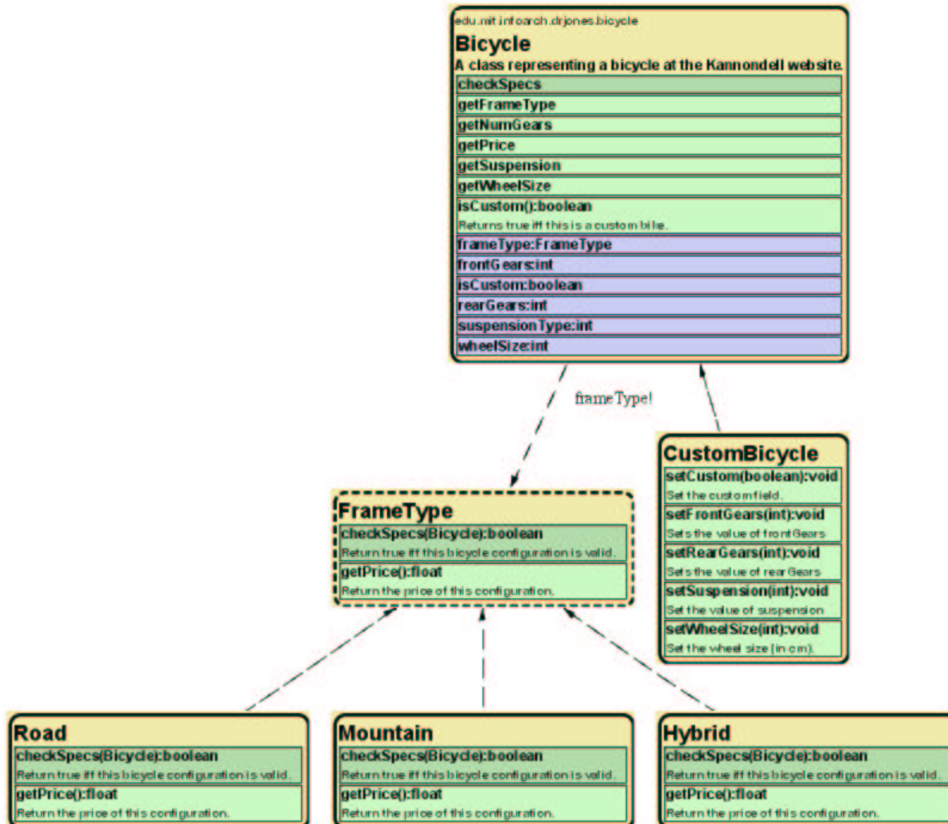

Figure 1-7: The second design alternative for the Bicycle class.

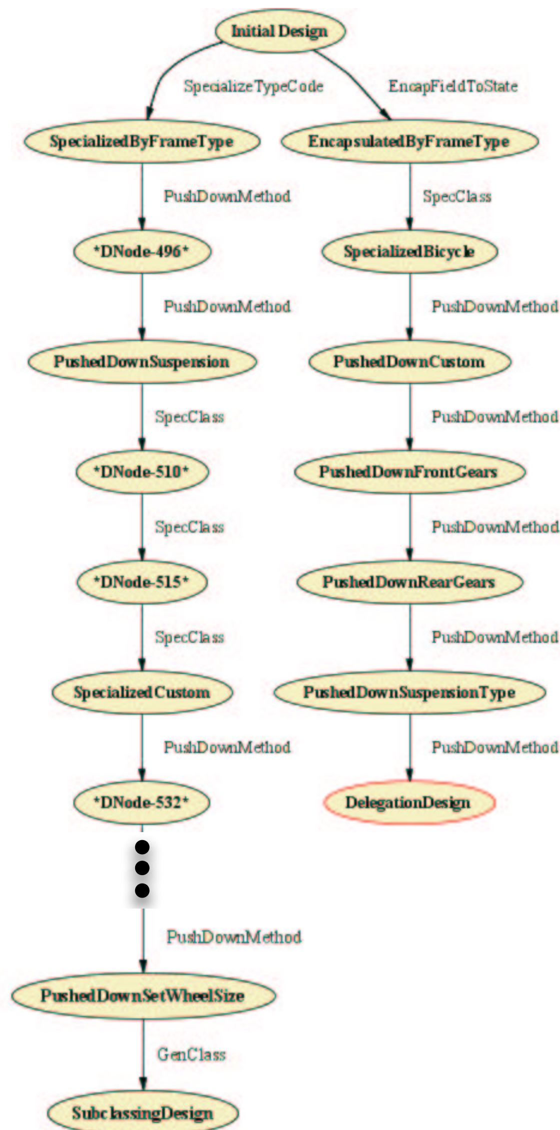

Figure 1-8: The map of the design space explored by Ecks.

space. When Ecks finds a satisfactory design, he can then turn to a source-level refactoring tool to implement it. This scenario illustrates the following key aspects of redesign with DR. JONES:

- DR. JONES gives the programmer the freedom to explore multiple design alternatives, each involving multiple refactorings, without having to transform the source at each step of the way.
- DR. JONES guides the redesign process by suggesting additional refactorings likely to improve the design. It also prevents refactorings which it knows will change the program's behavior in unintended ways.
- DR. JONES' crystal ball renders future designs visually, letting the programmer spot unforeseen opportunities for further design improvement.
- DR. JONES records the design space explored by the programmer – all of the design alternatives considered during a redesign session.
- The input to DR. JONES is an ordinary Java program, in source and compiled forms. DR. JONES' reverse engineering is self-contained and requires no additional programmer annotation.

1.5 Roadmap

Chapter 2 examines the process of software redesign in terms of the roles the programmer and the computer can play. It addresses the question, *What should a software redesign assistant do?* It compares the division of labor when redesigning with pen-and-paper, with a source transformation tool, and with DR. JONES. It motivates the claim that design exploration and source code transformation are tasks which can be dealt with separately.

Chapter 3 describe refactorings, which are the software redesign moves provided to the programmer by DR. JONES. It motivates the use of refactorings in DR. JONES, and compares their manifestation in DR. JONES with that in the Refactoring Browser (Roberts *et al.*, 1997). It also contributes a taxonomy of refactorings that organizes known refactorings and identifies new ones.

Chapters 4, 5, and 6 describe DR. JONES itself. Chapter 4 describes DR. JONES' knowledge base of refactorings, how the refactorings are related, and how the knowledge is used. It answers, *What does* DR. JONES *know about refactoring?*

Chapter 5 describes DR. JONES' design representation, which contains everything DR. JONES knows about the program. This representation has two interwoven parts, design instances and a design space. A design instance (DI) captures what DR. JONES knows about a particular program design. The design space (DS) captures which designs have

been generated during a session with DR. JONES and how they are related. It answers, *What does* DR. JONES *know about the program's design and its design space?*

Chapter 6 describes the dialogue between the programmer and DR. JONES. It describes how DR. JONES presents designs to the programmer, how the programmer proposes design changes and receives feedback, and how the programmer can navigate the design space by revisiting past design alternatives. It answers, *What can the user tell* DR. JONES*, and how does* DR. JONES *respond?*

Chapter 7 describes two redesign scenarios in detail. One is the Bicycle class, which was already outlined in this chapter. The other is the development of the JUnit unit testing framework through the repeated application of design patterns, as documented by JUnit's authors (Gamma and Beck, 2000). These scenarios are used to illustrate the strengths (and limitations) of DR. JONES' approach to design exploration.

Chapter 8 concludes the dissertation. It summarizes the thesis and contributions of the research. It presents related work and future work. The dissertation ends with a discussion of how the pervasive nature of software redesign should motivate the design of future programming languages, so that programs written in those languages become inherently easy to redesign.

Chapter 2

The Role of a Redesign Assistant

This chapter answers the question: *What Should A Software Redesign Assistant Do?* There are many possible roles for a software redesign assistant. I present the role chosen for DR. JONES and motivate it by comparing three toolsets for software redesign: pen-and-paper and a text editor, a source code refactory (that is, a tool that transforms source code to execute refactorings), and DR. JONES. The bicycle example introduced earlier will be used to illustrate how the various redesign tasks are shared by the programmer and his tools.

The comparison shows that the exploration of the program's design space – as distinct from transforming source code to execute design moves – is a critical activity in redesign. Among these three approaches, only DR. JONES provides computer support for design exploration. DR. JONES thus separates the concerns of design exploration and source code transformation, and addresses the real challenge for a redesign assistant: give the programmer support in the conceptually difficult part of software redesign, not the routine and mechanical execution of design moves. This separation of concerns recalls that of Fred Brooks' distinction between essential and accidental complexity in software engineering:

> *"I believe the hard part of building software to be the specification, design, and testing of [its design], not the labor of representing and testing the fidelity of its representation"* (Brooks, 1995)

Of course, DR. JONES is no silver bullet, but hopefully a step in the right direction.

We begin by presenting context for the task of software design, exploring what constitutes a program's design, and some of the reasons why software is redesigned.

The comparison among the three redesign approaches is then made in terms of the core tasks making up redesign: program understanding, flaw diagnosis, redesign planning, and redesign execution (i.e., source modification). I show which redesign activities are facilitated by which of the approaches. I also discuss how each approach results in a tradeoff with respect to automation versus programmer freedom.

The chapter concludes with a discussion of how a design exploration tool (like DR. JONES) and a source transformation tool can work together to enable rapid and reliable

redesign.

2.1 What Is The "Design" of a Program

It's helpful to begin by articulating what is meant by a program's "design." We assume it to be something more abstract than the program's source code, and believe that design exploration involves changes to this abstraction. However, if the distinction between design exploration and source transformation is to be clear, we should be precise about what information is included in the design abstraction.

All of the many decisions that go into the creation of a program could be considered part of its design, including:

- The choice of programming languages and environments;
- The user interface;
- Invariants that describe intended behavior;
- The choice of algorithms to satisfy those invariants;
- The naming and organization of the basic abstractions in the program.

Redesign can and does occur in all of these aspects. However, this research focuses on the last item in the list. For object-oriented languages like Java, this means the naming and organization of a program's classes, methods, and fields, including:

- The declarations of packages, classes, methods, and fields in the program;
- The containment relationships between those elements, i.e. which classes contain which methods;
- The inheritance (*is-a*) relationships between classes;
- The delegation (*has-a*) relationships between classes.

This information is typically captured in the object model of an object-oriented program (Booch *et al.*, 2001). Note that in the case of Java, nearly all of this information would still be available from a program if all of its expressions that compute values were removed. This sense of "design" is structural – it abstracts away the behavior of the program and the values it manipulates. Thus, although DR. JONES can redistribute behavior in the program (e.g., by moving methods), it cannot directly manipulate algorithms and invariants.

The list above summarizes the structures that can be viewed and manipulated by the programmer using DR. JONES. DR. JONES does maintain and use additional information about the program, primarily to help it provide suggestions and feedback (see Chapter

5). Even with this additional information, DR. JONES works with a representation of the program that is simpler than the original source code.

I have focused on the object model view of design for three reasons. First, the naming and organization of a program's abstractions indicates what the programmer feels is salient, and shows how he has broken down the design problem into solvable parts. The object model reflects some part of his mental model of the program. Other aspects of software design described earlier, while equally important, can all be stated in terms of the abstractions making up the object model. It defines the fundamental shape of the program.

Second, other maintenance activities like optimization, debugging, or adding new features are helped or hindered by the structures described by the object model. For example, it is simpler to maintain a program without circular *has-a* relationships (reflecting circular dependencies). Difficulties encountered during maintenance often motivate the redesign and refactoring of a program (Fowler, 1999), which restructures the object model and (hopefully) alleviates the difficulties.

Third, the object model is amenable to reverse engineering, because its contents are relatively explicit in the program (as compared to invariants). They can be recovered automatically and accurately without extensive programmer intervention (Waingold and Jackson, 1999).

From this point on, I use the term "program's design" to denote the content of its object model, unless stated otherwise.

2.2 Why Is Software Redesigned?

A brief survey of the forces motivating software redesign helps to establish some context of the discussion of specific redesign tasks to follow.

Why is software redesigned at all? Buggy or slow programs obviously need to be fixed, and a better design can make them easier to repair. But even a correct program may be unmaintainable; it becomes stuck at its location in design space, incapable of improvement or adaptation to meet new needs.

Because all software is written and maintained incrementally, it naturally exhibits a decay in structure over time (Griswold *et al.*, 1998; Griswold and Notkin, 1993; Belady and Lehman, 1985). Small changes disrupt the consistency of the initial design, barriers of abstraction are broken, and functionality becomes duplicated in several places in the program. In a larger sense, the current organization of abstractions no longer captures the best way to break down the current problem. This phenomenon manifests itself in a variety of ways.

- *Small changes are hard to make to the program.* The most common symptom is that the effects of a small change to the program cannot be predicted, or it becomes impossible even to find where to make the change. Structural decay spreads causes and

effects far apart in the program, and functionality crops up in unexpected places. These phenomena complicate the reasoning needed for software maintenance.

- *New features are difficult to add to the program.* It becomes hard to find a place for new functionality that fits with the program's architecture and maximizes reuse. A heuristic for spotting this symptom is to count how many places the existing code has to change to add the new feature.

- *The program is solving the wrong problem.* The difficulty of gathering correct and complete requirements means that often the initial version of a program does not meet the needs of those who will use it. And, even good requirements gradually become less and less relevant over time as the demands and the environment of the software evolve. Program designs must constantly adapt to match the problem they are trying to solve.

- *In hindsight, a better design is possible.* Even if the requirements are well understood, the best design to meet them is rarely apparent at the outset. Instead, designs evolve incrementally along with the software, and often the best abstractions for the program emerge only when it is nearly finished.

Each of these forces weighs into the equation for deciding when to do redesign. Software redesign certainly has a cost, but not doing redesign when needed has a higher cost in the long run.

2.3 Software Redesign Tasks

Redesign involves understanding the program, diagnosing its design flaws, planning how to mitigate those flaws, and executing the plan by modifying the source code. These tasks form a cycle of redesign activity shown in Figure 2-1, with the information gathered and decisions made at each step feeding forward into the next. Figure 2-1 simplifies reality, however, because in practice programmers jump around from task to task and gain program understanding throughout the redesign process.

These tasks can be distinguished by the kind of knowledge and expertise required to perform them. Understanding a program requires knowledge of its programming language and environment as well as its application domain. Diagnosing and fixing design flaws requires knowledge of good and poor design practice, as well as methods to remedy design flaws. Executing design changes requires knowledge of the programming language and its semantics, as well as how to use tools to modify source without introducing syntactic or semantic errors.

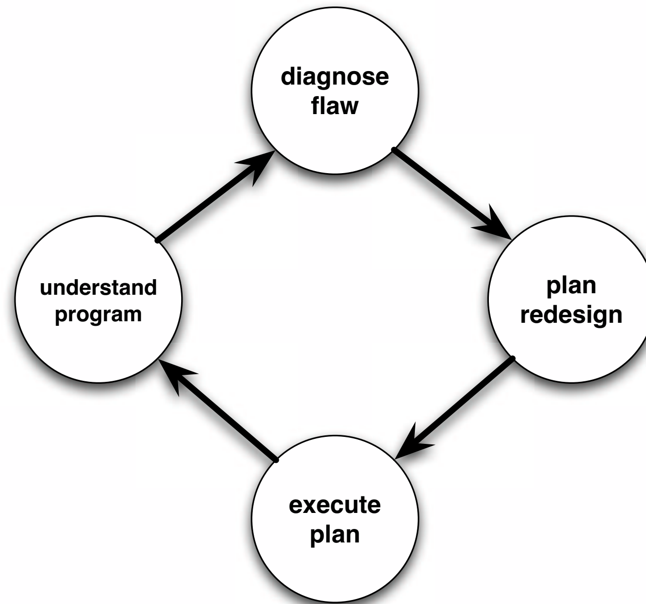

Figure 2-1: A model of the tasks in software redesign.

2.3.1 Understanding the Program

All redesign begins and ends with program understanding. Program understanding is any activity that adds to the programmer's knowledge of the program's structure, behavior, and environment, and how they all relate to the programming problem. To gain this understanding, the programmer reflects on his own experiences with the program, reads its documentation and source, and consults other programmers. The programmer's resulting mental model integrates all of his accumulated experiences with the program (Storey *et al.*, 1997a; von Mayrhauser and Vans, 1995). Program understanding is the central activity in software redesign, because it influences all of the programmer's judgments about if, where, and how to redesign.

External resources and tools play a critical role in supporting this activity. Programmers use pen and paper to take notes, annotate printouts, and sketch diagrams. A variety of computer tools (both research prototypes and commercial products) support program understanding through reverse engineering and visualization (Eick *et al.*, 2002; Systä, 1999; Walker *et al.*, 1998; Storey *et al.*, 1997b).

2.3.2 Diagnosing Design Flaws

Diagnosis is the identification of specific design flaws in the program. The existence of a flaw does not imply that a mistake was made in the original design. Instead, an aspect of the original design has become an impediment to desired changes to the software. An examples of a flaw is the choice of an integer type code to represent a frame type in the

Bicycle class. It worked well enough initially, but became unwieldy when the class was faced with new demands.

Sometimes, antipatterns or bad smells can be recognized in the program (Beck and Fowler, 1999). These are design flaws which recur repeatedly in a variety of programs and circumstances, and have been generalized and named. Examples of bad smells include ClassTooLarge, FeatureEnvy, and LongParameterList. The development of a catalog of bad smells is encouraging, since tools can use the catalog to help the programmer find them in their programs (Florjin, 2002). Of course, every program has its idiosyncratic flaws, too.

Traditionally, programmers spend a great deal of effort on finding the "perfect" program design up front, before implementation begins. Some practitioners have abandoned this view and treat a design as inherently flawed; it must be incrementally developed and continually debugged, just like the program implementing it (Beck, 2000). This approach echoes Sussman's comment that "programming is debugging a blank sheet of paper."

Note that diagnosing a flaw, by itself, often gives little indication of how to fix it. A class that is too large offers a large number of possibilities for its decomposition.

2.3.3 Planning Redesign

Redesign planning is the task of choosing how to eliminate the design flaw from a myriad of possibilities. This is the most difficult part of redesign, because not only must the flaw must be eliminated, but correct behavior maintained and new flaws not introduced. Knowledge of how the program may change in the future often guides the plan; for example, it doesn't make sense to combine two classes that may each be specialized later. The programmer's experience, expertise, and judgment come into play, and tools that assist planning must collaborate tightly with the programmer.

A plan trades off effort and disruption now for less effort in the future. However, the future direction of the software is typically unclear. The programmer is always guessing about whether his changes will help or hinder him in the future. The ability to visualize future program designs would be invaluable; it would allow the programmer to better judge the impact of his proposed changes.

In general, a plan may transform the program arbitrarily, possibly adding and removing functionality. I focus on plans that preserve the existing functionality of the program. If the plan preserves the overall behavior of the program, it is possible to break it down into simpler moves called refactorings, as discussed in Chapter 3. DR. JONES lets the programmer formulate plans as sequences of refactorings.

2.3.4 Executing the Plan

Once a course of action has been chosen, the programmer turns to the source code to execute it. The amount of effort involved at this step depends on the sophistication of his tools.

With a basic text editor, the programmer uses cut/copy/paste and search-and-replace operations to rearrange code and change names. This is a tedious and error-prone operation, and some errors may not be caught by the compiler[1]. If the programmer is lucky, he has a tool which understands the structure of the source code and safely automates much of the code transformation.

2.4 Dividing The Labor: Building Bicycles Revisited

This section compares the roles of three toolsets for redesign: pen-and-paper and a text editor, a source code refactory, and DR. JONES. In particular, like pen-and-paper, but unlike source refactories, DR. JONES supports the conceptual tasks of redesign. I use the Bicycle example from chapter 1 to compare these tools.

2.4.1 Pen-and-Paper and a Text Editor

Let's return to Ecks Presso's situation, assuming he has only basic tools: pen and paper, and a text editor. He's just come out of a meeting and realizes his Bicycle class won't support all the new site features that were proposed. Since he hasn't looked at that class in a while, he first needs to get an idea of what it does. Grabbing a notepad, he browses the source and draws something like Figure 2-2. He notes on this diagram the design flaws he discovers and needs to fix.

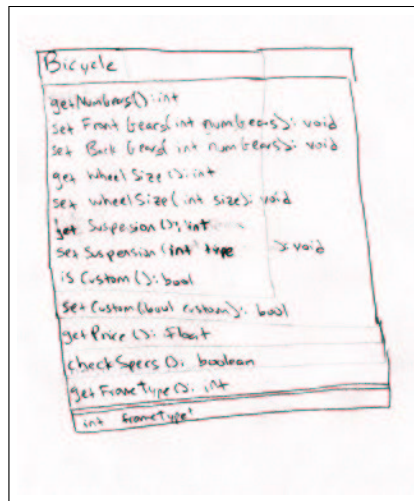

Figure 2-2: A hand-drawn diagram of the Bicycle class.

With this diagram, he considers how he might change the design. He draws additional diagrams that capture different points in the design space, like those in Figure 2-3. These

[1] Such as failing to rename an entity which happens to be captured by an enclosing scope.

diagrams are useful because they are concise, capturing the essential details of the design. However, they are static; they don't give feedback about the wisdom of his proposed changes, and details of the designs aren't available, unless he takes the extra time to draw them in. And, unless he plans out his changes step by step, he will have to construct a detailed plan for modifying the code separately, or make it up "on the fly."

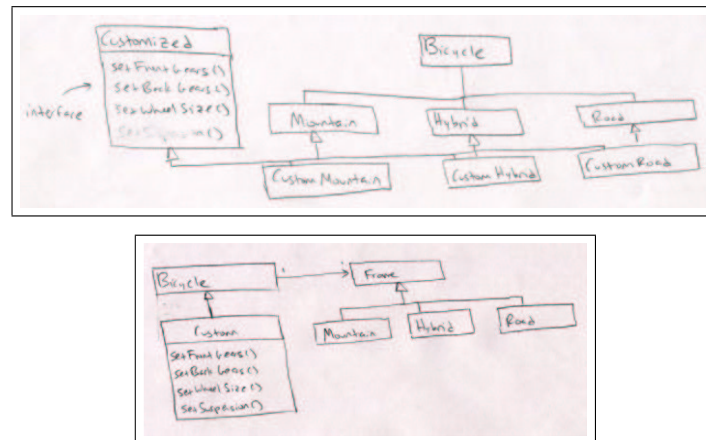

Figure 2-3: Design alternatives that Ecks would draw with pen and paper .

Once he finds a satisfactory new design, he turns to a text editor like Emacs to execute his changes. Some of his changes involve the relocation of existing code, using Emacs' cut and paste operations. Ecks must be careful; a misstep that introduces a syntax error could lead to a frustrating compile/re-edit/compile cycle. Even more troubling, he could make a change that is syntactically correct (and compilable) but introduces a bug into the program. Such mishaps can be caught with a good suite of test cases, but it would have been better to not have introduced the bug in the first place.

Also, he may decide in the middle of his modifications that his new design wasn't the best alternative after all. He then faces two choices: revert to the original version and start from scratch, or keep the new design and hope for the best. With such basic tools, the time and effort needed to complete one iteration of the redesign cycle discourages design exploration.

Figure 2-4 summarizes the role of the computer in pen-and-paper redesign. (The icons indicate which tasks are manual and which tasks are computer-supported.) The conceptual aspects of redesign – program understanding, diagnosis, and planning – are entirely up to the programmer. The computer provides a certain degree of support for modifying the program text. This situation maximizes freedom for the programmer, at the cost of extra effort, long redesign iterations, and the risk of unintended changes to the program. Despite these shortcomings, pen-and-paper redesign remains a common choice for programmers.

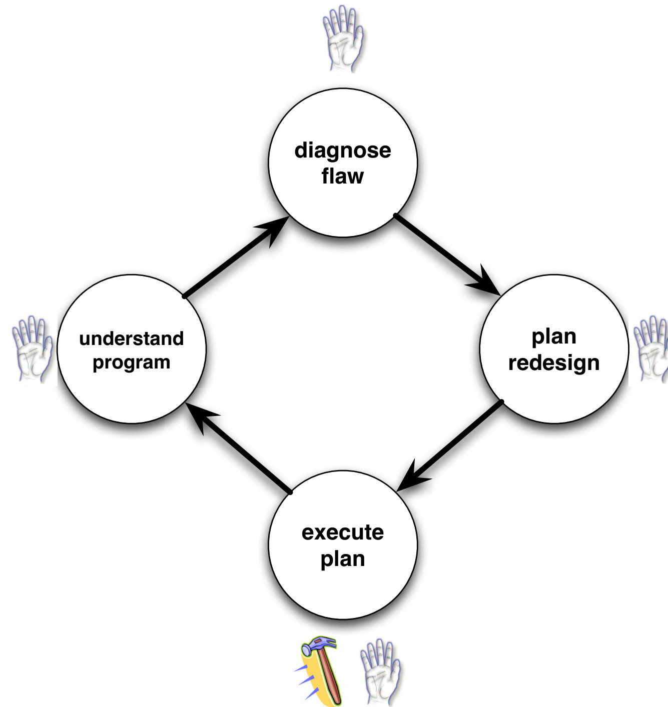

Figure 2-4: With pen-and-paper redesign, the computer offers limited support for design evolution.

2.4.2 Source Code Refactories

Given a refactoring, a source code refactory automates its execution by transforming the program's source code. An early example of this kind of tool is the Refactoring Browser for Smalltalk (Roberts *et al.*, 1997). More recently, commercial development environments for Java have begun to incorporate automated refactorings.

Suppose Ecks had such a refactory at his disposal. The refactory would be of enormous benefit to the Ecks; it removes much the risk associated with refactoring. However, it can't replace the pen and paper diagramming that Ecks uses to plan which refactorings to do. Thus, most of the benefit accrues after Ecks has already made the decisions about how to redesign the program. Three aspects of Ecks' interaction with a refactory support this claim.

- *A refactory works at the source level.* Interaction with a typical refactory is at the level of individual lines of code. For example, to push down the getSuspension() method in Bicycle, Ecks would have to navigate the Bicycle.java file to find the declaration for getSuspension(), and then give the refactoring command. Once given, the tool asks him to confirm each place in the source that is changed as a result. This source-level view requires a longer interaction to complete one redesign iteration, and interrupts the programmer with source-level details when he would rather be thinking about design. It lacks the freedom of pen-and-paper redesign, where the programmer can

work at the level of design moves, not source code changes.

- *A refactory lacks design assistance.* A refactory may check that refactorings are safe (behavior-preserving), but doesn't provide guidance to help the redesign process. At best, it gives a yes-or-no answer indicating if a refactoring is allowed.

- *A refactory doesn't maintain a design space.* Although Ecks might be able to create his two alternative designs with the refactory, he would have to create two entire versions of the source code to do so. And, the refactory wouldn't help him switch between the two alternative designs; he would have to use an separate source code control system to check out the other version, and then resynchronize his workspace.

The situation is summarized in Figure 2-5. It leaves the programmer in better shape than pen-and-paper design by reducing the effort required to execute design changes. However, all the important design decisions are made in the understanding, diagnosis, and planning steps, steps where the refactory does not play a role.

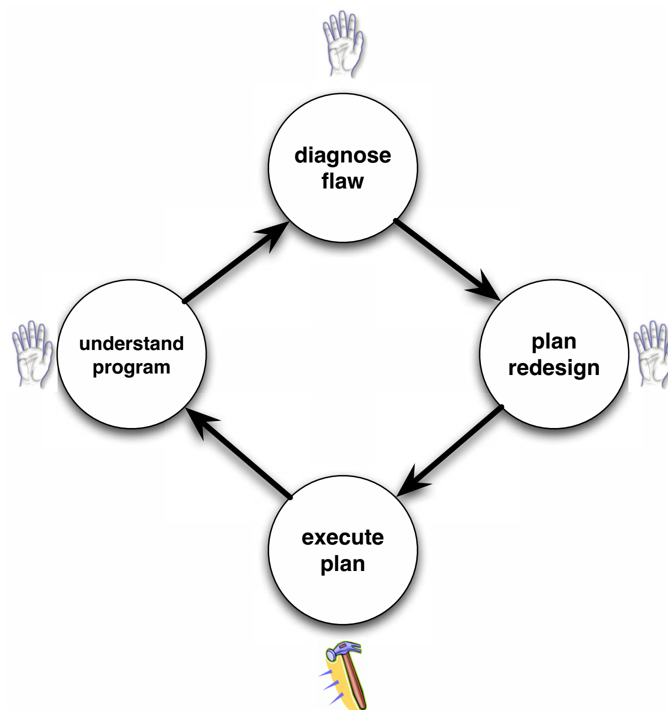

Figure 2-5: Source code refactories benefit the programmer after design decisions have been made.

2.4.3 Dr. Jones

DR. JONES is intended to help the programmer with design exploration. It plays the role of a smart colleague who knows a lot about programs and refactorings, but little about what

a specific program does. It helps the programmer see the current state of the program's design, and explore design alternatives from that point. DR. JONES attempts to combine the freedom of design exploration with pen-and-paper, with the assistance of a colleague who does the diagramming, bookkeeping, and checking for you. This freedom is essential for the rapid, critical, and reflective nature of design exploration.

With DR. JONES, Ecks can plan his redesign apart from its execution. DR. JONES emulates the role of pen and paper, but with the added benefit of the computer's assistance. DR. JONES differs from a source code refactory in three important aspects:

- DR. JONES *hides source level details from Ecks.* Ecks can work with the essence of the design, and step back from its source level details. This ability saves time and effort over pen-and-paper diagramming, but more importantly, it automatically focuses the programmer's attention on the salient structures and abstractions in the program. DR. JONES' diagrams have a dramatically higher density of design elements versus source code; for example, compare the 30-50 lines of text visible in a source editor (enough for a few methods) with the multiple classes and relationships depicted in Figure 1-6.

- DR. JONES *assists Ecks as he redesigns.* DR. JONES knows how refactorings relate to one another from a design perspective. Like a source refactory, it analyzes the program to check if a refactoring preserves behavior. But, it also warns Ecks if a refactoring introduces a design flaw, and suggests additional refactorings that lead to better designs.

- DR. JONES *maintains the design space explored by the programmer.* It keeps a history of the designs considered by the programmer, and the refactoring moves that led the programmer to each of them. Ecks can switch between the plausible alternatives in Figures 1-6 and 1-7 with a single mouse click, and revisit intermediate points in the design space as well.

DR. JONES' role in the redesign process is summarized in Figure 2-6. DR. JONES focuses on the three conceptual tasks in redesign. It assists program understanding by extracting and visualizing design-level information. It assists flaw diagnosis by diagramming current and future designs, which exposes flaws that would be difficult to see otherwise. And it assists planning, by capturing the programmer's intended redesign moves, keeping track of the design alternatives explored, and visualizing future designs.

2.5 Freedom Versus Power

Another way to compare toolsets for redesign is to place them on in a space plotting the freedom versus the power they afford the programmer, as in Figure 2-7. These two attributes must generally be traded off by a redesign toolset. Giving the programmer more

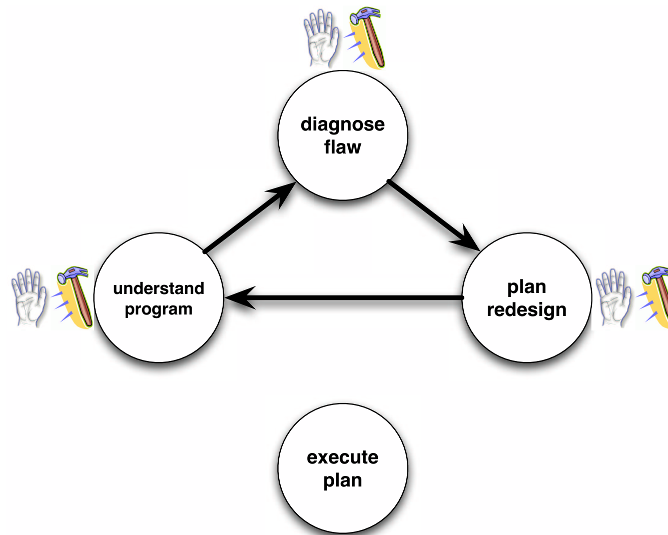

Figure 2-6: DR. JONES lets the programmer rapidly complete design iterations without manipulating the source code.

freedom in redesign sacrifices some of the support the tool could provide, because no tool can automatically adapt itself to every programmer and every program. On the other hand, restricting the programmer to redesign in certain ways limits the programmer, but gives the tool more of an opportunity to share in the effort. DR. JONES tries to find a middle ground in this space, a part of the space currently unoccupied by any other toolset.

Examining toolsets and the points they occupy in the freedom-versus-power space illustrates the tradeoff further. At one extreme is pen and paper. With pen and paper, the programmer has ultimate freedom to redesign the software as he wishes. He is not restricted by any notation, any predetermined set of design moves, or the constraints inherent in the current design. Because of the complexity of the task at hand, most programmers desire this level of freedom, so they can redesign in a way that suits their style and the problem and hand.

The downside is that the computer can play only a limited role in the process. The programmer can certainly use it to generate printouts or diagrams of the source code, or browse the program while redesigning. But, the redesign itself happens away from the computer, and so this toolset has limited power in assisting the programmer.

At the other extreme are source refactories. These tools are very powerful, by automating the tedious and precise work of applying refactorings to source code. However, they remove much of the freedom of design exploration with pen-and-paper. The programmer is restricted to the refactorings implemented in the tool. While refactoring, he must attend to source-level details, for example by confirming source modifications. There is a cost to undo an action, and exploring multiple design alternatives involves managing multiple versions of the source code. Further refinement of these tools could overcome many of

these problems, but even then, their additional power is of most help after the programmer has decided how to change the program.

DR. JONES occupies a point between these two extremes, acting as a "smarter pen and paper." Like pen and paper, it offers the freedom of working with a design, without touching the source code. It offers additional power by visualizing current and future designs and providing concrete design assistance to the programmer. There is a tradeoff for this power, in that the programmer is restricted to use a fixed set of refactorings, but the set is intended to be broad enough to support meaningful design exploration.

As a final point of comparison, consider a tool that allowed the user to create and edit UML diagrams. With such a tool, the programmer is free to produce any new design within the constraints of the UML notation. However, without analysis of the program and the design advice derived from it, the only real power is in rendering neater diagrams of future designs that may not preserve desired functionality.

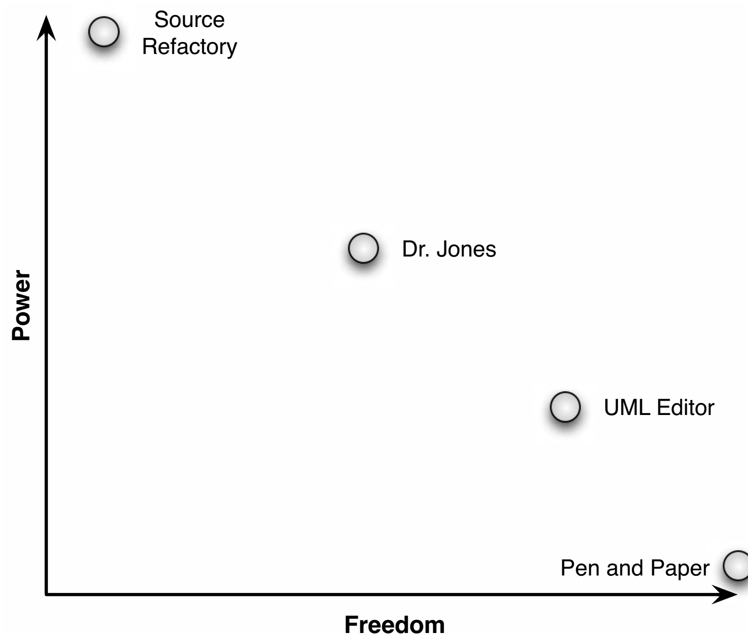

Figure 2-7: A qualitative comparison of the freedom versus power afforded by toolsets for redesign.

2.6 Summary

Software redesign consists of design exploration and source modification. DR. JONES assists the former; source refactories assist the latter. To answer the opening question of this chapter, a software redesign assistant should ideally combine these capabilities. The end result would be a tool that focuses the programmer's effort on the conceptually difficult aspects of software redesign, and automates the mundane and repetitive activity of source

code transformation. Such a tool would enable the rapid and reliable exploration of a program's design space.

Chapter 3

Redesigning With Refactorings

When programmers collaborate to redesign a program, they communicate through a shared vocabulary of design moves. This should also hold true when a programmer is working with the computer: the programmer should be able to describe concisely and precisely how he wants to change the software, and have his intentions understood. The partner (computer or human) can then offer criticism, alternatives, or act on the program to realize the design intention.

This chapter will present the vocabulary of refactorings chosen as the redesign language for Dr. Jones. Refactorings are program transformations that preserve the overall behavior of the program, but can improve its design dramatically. Refactorings form a good redesign vocabulary because they are simple, well-known, and well understood through prior research.

Dr. Jones acts on this vocabulary through a refactoring knowledge base. This knowledge base is particularly suited for interactive redesign, and so contains additional knowledge about refactorings not found in source refactoring tools. I compare the content and character of the knowledge in Dr. Jones and the Smalltalk Refactoring Browser (a source refactoring tool (Roberts *et al.*, 1997)) to illustrate Dr. Jones' additional knowledge.

I also develop a taxonomy to organize known refactorings and identify new ones. The taxonomy is based on the metaphor of a refactoring as a natural language command that combines an action verb with a context of operands. I categorize fifty-two refactorings with a novel set of thirteen such verbs. By interpreting a refactoring verb in new contexts, new refactorings can be systematically identified, instead of being discovered through practice.

3.1 Refactorings: An Overview

3.1.1 Refactorings Are Local, Structural Changes

Refactoring is a systematic way of reorganizing existing software. The reorganization process is composed of simpler steps called refactorings. Each refactoring makes a small

change to the structure of the program, but the cumulative effect of many refactorings can be profound. Examples of refactorings include renaming a method, moving a method from one class to another, and extracting a base class from two similar classes.

According to Martin Fowler, a refactoring is a

> *a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior* (Fowler, 1999, p. 53).

Three aspects of this definition stand out. First, refactorings are *local* changes; only a small group of related elements are targeted. Second, refactorings preserve behaviors visible to those who use the program (although intermediate values and states may change). Third, because refactorings are behavior-preserving, they are easier to automate. This is because a refactoring tool doesn't have to know what the program is trying to accomplish; it only has to ensure that the program will produce the same final values before and after the refactoring.

Refactoring while preserving behavior does not always mean keeping the same set of methods. For example, removing a method that is not currently called, strictly speaking, does not preserve all of the program's behavior. Also, adding a new, unreferenced method to a class expands the class' possible behaviors. However, these are considered valid refactorings when the overall functionality of the program is maintained.

Refactorings are simply program transformations, and do not inherently improve the design of the program. In fact, if applied arbitrarily they will almost certainly degrade the design of the program. The key to refactoring is applying them judiciously, with the primary goals of refactoring in mind: repairing specific design flaws, increasing the clarity of the resulting program, and simplifying other maintenance tasks. They are one tool for software maintenance, to be used in the context of other redesign processes, like program understanding, flaw diagnosis, and repair planning.

3.1.2 Refactorings Are Dr. Jones' Redesign Language

There are several reasons why refactorings were chosen as the redesign vocabulary for DR. JONES.

- Refactorings have the properties of a good vocabulary. They are self-descriptive, simple and can be composed. A sequence of simple refactorings can make a large move through a program's design space. Even the complex task of applying a design pattern involving many classes to an existing program can be broken down into refactorings (Tokuda and Batory, 2001). Later, the development of the JUnit unit testing framework with DR. JONES is used to illustrate this situation.

- Refactorings are accepted software engineering practice. Many programmers have been exposed to them through their education or experience. Their integration into

recent commercial development tools reflects their growing acceptance. A programmer who knows refactorings can approach a tool that uses them with expectations about what they mean and how they will affect his program.

- Refactorings are fairly well-understood. Many years of research into refactoring software has produced both formal descriptions of refactorings (Opdyke, 1992; Roberts, 1999), tools that automate them (Roberts *et al.*, 1997; Moore, 1995), and discussions of their design implications (Johnson and Opdyke, 1993; Cinne'ide, 2000). This research builds on this previous work to extend refactoring knowledge and applications.

Using refactorings as a redesign language has some drawbacks as well. First, a given refactoring is not always applicable to a given program. Each refactoring carries with it a set of preconditions that determine when the refactoring will preserve behavior, and will not introduce syntax errors or bugs. In some cases, the applicability of two or more refactorings is order-dependent. Evaluating these preconditions and handling situations when they fail adds to the complexity of refactoring tools (such as DR. JONES).

Refactorings are not as well understood for programming languages with higher order functions or parameterized type systems (like ML or Haskell), although there is some recent work in this area (Li *et al.*, 2003). Java does not currently have these features, but is evolving in that direction (Bracha *et al.*, 2001).

Also, refactorings are primitives – they represent a family of the simplest meaningful design moves. A programmer's design intentions are likely to be formulated at a higher level, such as "remove the dependency between these two classes." Pursuing such a higher-level goal requires the programmer to select and execute many individual refactorings. Ideally, tools should help the programmer break down high level redesign goals into refactoring steps, as well as let him apply individual refactorings.

Finally, there is no universal redesign language for software. Every programmer has his own variations on refactorings and his own bag of redesign tricks. Programmers and their partners develop a dialogue that is idiosyncratic and specific to the problem at hand. Ideally, a programmer and his tools should adapt to each other in the same way, like an experienced programming pair.

Even with these drawbacks, refactorings represent a simple and well-understood set of redesign moves in object-oriented programs. They form a common vocabulary for redesign collaboration between the computer and the programmer.

3.2 Refactoring Knowledge

A large part of the power of refactorings is that they can be automated. Tools that do so must contain knowledge of the refactorings they automate. The character and content of that knowledge determines the tool's focus and capabilities. Current tools are *source refac-*

tories, which focus on checking whether a refactoring will preserve the program's behavior, and transforming the program text to execute it.

A tool that supports design exploration by refactoring must have a different but overlapping set of knowledge. It should know if a refactoring preserves the program's behavior, but also know how the refactoring impacts the program's design. With this additional knowledge, the tool can provide assistance at the design level. The following comparison of the knowledge in the Refactoring Browser and DR. JONES illustrates the differences in the two sets of knowledge.

3.2.1 The Knowledge in the Refactoring Browser

The Refactoring Browser for Smalltalk is a source refactoring tool for Smalltalk programs (Roberts *et al.*, 1997). It contains three kinds of knowledge about its eighteen refactorings: preconditions, source transformations, and postconditions.

- **Preconditions.** These are boolean conditions on the program that must be satisfied to guarantee that the overall behavior of the program won't change after the refactoring. Many of these conditions, such as name conflicts, are relatively simple to check. Other conditions are more difficult, and require proving more difficult facts about the runtime behavior of the program.

  An example of a precondition in the Browser is the one for the RenameClass(oldName, newName) refactoring:

  $$IsClass(oldName) \wedge \neg IsClass(newName) \wedge \neg IsGlobal(newName)$$

  This expression can be read, "If oldName names a class, and no other class or global variable is named newName, then the refactoring is legal."

- **Source Transformations.** These are modifications to the text of the program that execute the refactoring. These are usually expressed as patterns and actions that rewrite parts of the abstract syntax tree (AST) of the program. In Smalltalk (like Java) more than one method or variable can have the same name; thus simpler string substitution cannot be used, because it cannot resolve those sorts of ambiguities.

- **Postconditions.** The Browser includes refactoring postconditions in its knowledge base. Postconditions are assertions that must be true after a refactoring succeeds. One of the postconditions for the RenameClass refactoring is

  $$\neg IsClass(oldClass) \wedge IsClass(newName).$$

One can use the postconditions of one refactoring to help satisfy the preconditions of the next. This reduces the amount of program analysis required, makes explicit the dependencies among refactorings, and thus allows a tool (in principle) to plan sequences of refactorings that are guaranteed to succeed.

The Refactoring Browser uses these three kinds of knowledge in the following way:

1. The programmer selects program elements and a refactoring to perform upon them.
2. The Browser analyzes the program and evaluates the refactoring's preconditions to verify that behavior would be preserved. If any precondition fails, the programmer is informed with an error message and the refactoring is aborted.
3. The Browser transforms the program source, and recompiles the methods which have changed.
4. The Browser asserts the refactoring's postconditions to update its stored analysis of the program.

3.2.2 The Knowledge in Dr. Jones

The Browser contains knowledge sufficient to transform the program source safely, but the interaction between the programmer and the refactory is one-way: the programmer specifies the refactoring, the Browser goes away to transform the code, and then awaits the next command. Moreover, if a refactoring cannot be applied, it is up to the programmer to devise an alternative. A design exploration assistant should have a more interactive dialogue with the programmer, and therefore must have additional kinds of knowledge. DR. JONES has five kinds of knowledge about each refactoring: must-guards, should-guards, suggestions, design transformations, and source edits.

- **Must-guards.** Must-guards are analogous to preconditions in the Refactoring Browser. These are boolean conditions on the current design that prevent refactorings from changing the program's behavior. Unlike the Browser's preconditions, however, some of the must-guards have repairs. A repair is an action suggested to the programmer when a condition is unsatisfied. By performing the repair, the programmer can satisfy the conditions of the initial refactoring, and permit it to proceed.
- **Should-guards.** Should-guards are used to warn the programmer that a refactoring would introduce a design flaw. Like must-guards, they are boolean conditions on the current design, and can have repair suggestions attached. Unlike must-guards, DR. JONES allows the programmer to ignore them and proceed with the refactoring anyway. An example of a should-guard is, *Don't create a field so that a field in a subclass would have the same name.* In this case, although the semantics of the program

would remain the same, an undesirable ambiguity would be introduced that makes the program harder to understand.

- *Suggestions.* After a programmer makes a refactoring, there may be related refactorings that make good design sense in that context. For example, after moving one method one should also move its overloaded variants – those with the same name but different argument types – since they probably represent similar behaviors. For each refactoring, DR. JONES has a list of additional refactorings, which are conditionally suggested depending on the new state of the design.

- *Design Transformations.* DR. JONES has a design transformation for each refactoring, analogous to the source transformation in the Refactoring Browser. This transformation updates DR. JONES' representation of the program's design. This design representation is used to render accurate diagrams of future designs, as well as evaluate must-guards, should-guards, and suggestions. Because the design representation is simpler than the source from which it was derived, design transformations are generally simpler than corresponding source transformations.

- *Source Edits.* Although DR. JONES does not modify the source of the programs it redesigns, each refactoring has templates for plain English instructions on how to edit the source to execute the refactoring. As refactorings are made, these templates are substituted appropriately and the results added to a list of editing instructions. In principle, the programmer could use the instructions to guide manual source refactoring after a session with DR. JONES. In reality, this capability would be extended to interface with a source refactory to automate source transformations.

The two lists of knowledge reveal the differing foci of the two tools. The Refactoring Browser focuses on the verification and execution of refactorings, while DR. JONES focuses on maintaining a design dialogue with the programmer. For example, DR. JONES has repair actions for unsatisfied guards, so that programmers are less likely to get stuck while redesigning. DR. JONES also makes suggestions so that some of the logical steps implied by a refactoring are readily available to the programmer. Finally, DR. JONES' design representation is simpler than the source code. Refactoring transformations are easier to specify in this representation, and it facilitates the expression of DR. JONES' other kinds of knowledge.

3.3 Organizing The Space of Refactorings

Breadth as well as depth of knowledge determines the utility of a refactoring tool. DR. JONES' goal is to give programmers much of the freedom of pen-and-paper redesign, where any design move is possible. Therefore, it is important that DR. JONES provide a broad and useful set of refactorings for the programmer.

Prior efforts have focused on accumulating the refactorings found useful by practicing programmers. An example of this approach is Martin Fowler's book *Refactoring* (Fowler, 1999), a catalog of 72 refactorings used by the book's author and contributors. Fowler groups them into chapters based on the general objective of the refactoring ("composing methods," "organizing data"), but does not offer a more general organization scheme.

This research contributes a way to organize many of the known refactorings and discover new ones. The fundamental idea is to start with the generic actions common to many refactorings, and evaluate those actions in different contexts. The context of such a generic action is the program elements it can act upon. We can form a space of potential refactorings by combining a set of actions with a set of potential contexts. Not every point in this space is sensible, but many correspond to useful design moves.

For example, consider the Compose action, which takes two elements of the same kind and combines them. In Java, it makes sense to compose packages (by combining their classes), classes (by combining their fields and methods), and methods (by combining their bodies), but not fields and parameters, which lack parts that can be combined.

Create	Create a new, empty program element.
Remove	Remove an existing element which is empty or unreferenced.
Rename	Rename an element.
Move	Move an element from one container to another.
Hide	Decrease (make more private) the accessibility of an element.
Reveal	Increase (make more public) the accessibility of an element.
Encapsulate	Introduce a new element to enclose an existing element.
Expose	Remove the container enclosing an element.
Compose	Combine the contents of one element with another.
Decompose	Extract part of an element to form a new element.
Generalize	Make an element more general in terms of the inheritance hierarchy.
Specialize	Make an element more specialized in terms of the inheritance hierarchy.
Alter Type	Alter the type of a field or parameter.

Table 3.1: A set of refactoring verbs that correspond to generic refactoring actions.

Each of the actions on one axis of the refactoring space is called a *refactoring verb*. The thirteen refactoring verbs used in the space are defined in Table 3.1. As potential contexts, we take each of the major kinds of program elements in Java: packages, classes, methods, fields, and parameters. Combining the verbs with the contexts creates the refactoring space as shown in Table 3.2. Not all of the points in this space make sense, and some (like specializing a class) could have more than one interpretation depending on the programmer's exact intention. (For these cases, the number of interpretations is shown in parenthesis.)

Overall, however, the majority of points in this space (fifty-two of sixty-five) map onto a meaningful refactoring.

The term *refactoring verb* is used because the combination of an action with a context suggests a natural language utterance that describes it. For example,

$$verb = \mathsf{Compose}, context = (\mathsf{Class}_1, \mathsf{Class}_2)$$

could be read as “Compose Class_1 with Class_2.” This correspondence was intentional, as it is easier for a tool to describe refactorings to the user this way, and also easier for the user to describe refactorings to a tool (if it were given an appropriate parser).

	Package	Class	Method	Field	Parameter
Create	√	√	√	√	√
Remove	√	√	√	√	√
Rename	√	√	√	√	√
Move	√	√	√	√	√
Hide		√	√	√	
Reveal		√	√	√	
Compose	√	√	√		
Decompose	√	√	√		
Encapsulate		√	√	√ (3)	√
Expose		√	√	√	
Generalize		√	√	√	√
Specialize		√ (2)	√	√	√
Alter Type				√	√

Table 3.2: The refactoring space, with points checked that are meaningful refactorings.

	Package	Class	Method	Field	Parameter
Create	**S**	**S**	**B**	**S**	**B**
Remove	**S**	**S**	**B**	**S**	**B**
Rename	**S**	**S**	**B**	**S**	**S**
Move	**S**	**S**	**B**	**B**	**S**
Hide		**S**	**B**	**S**	
Reveal		**S**	**S**	**S**	
Compose	**S**	**B**	**S**		
Decompose	**S**	**B**	**B**		
Encapsulate		**S**	**B**	**B**	**B**
Expose		**B**	**S**	**S**	
Generalize		**B**	**B**	**B**	**S**
Specialize		**B**	**B**	**B**	**S**
Alter Type				**S**	**S**

Table 3.3: The overlap between the refactoring space and *Refactoring*.

The refactoring space approach yields two benefits. First, it enumerates the complete

set of refactorings within the scope of actions and contexts. If the actions and contexts are sufficiently broad, the refactorings found in the space are likely to be an expressive and useful set of design moves. Second, the space can identify refactorings not mentioned in existing catalogs (Roberts *et al.*, 1997; Fowler, 1999). These additional refactorings widen the programmer's vocabulary of design moves in all circumstances, whether he is using a tool or pen and paper.

Table 3.3 compares the refactorings in the refactoring space against those in *Refactoring*.[1] Refactorings marked with **B** are found in both the refactoring space and *Refactoring*, and those with **S** are found only in the refactoring space. The overlap between the two sets is significant: the space covers thirty of the seventy-two refactorings cataloged in *Refactoring*. In addition, the refactoring space approach identifies twenty-one additional refactorings not included in *Refactoring*.

Refactoring does include forty-two refactorings not found the refactoring space. Of these, twenty-three target expressions in method bodies, which DR. JONES does not do. Fifteen more are composite refactorings, in the sense that they could be accomplished by composing two or more refactorings in the refactoring space. The remaining four simply do not fit neatly into the refactoring space. They are "Introduce Foreign Method," "Change Reference to Value," "Change Value to Reference," and "Encapsulate Downcast." These are all useful design moves, and the programmer should not be limited to those that can be expressed in the refactoring space (or any similar framework). However, the refactoring space does capture a wide range of useful refactorings at the design level.

Also, the refactoring verbs are not very particular to refactoring Java programs. Abstractions from some other programming language (or even a database schema) could be placed in the columns of the refactoring space, and useful refactoring possibilities would emerge. The refactoring space is a step towards the systematic enumeration of design transformations in a variety of domains.

3.4 Summary

This research adopts refactorings as the redesign vocabulary underlying the interaction between the programmer and DR. JONES. Refactorings are simple, local design moves that preserve a program's overall behavior. They are accepted software engineering practice and are well-understood from a research viewpoint. DR. JONES provides twenty-two such refactorings to its user.

This research contributes to work on refactoring:

- A knowledge base of refactorings created from the perspective of a program redesign assistant. This knowledge base contains refactoring knowledge particularly suited

[1] Consult Appendix A for a detailed list matching refactorings in the refactoring space to those in *Refactoring* as well as the Smalltalk Refactoring Browser.

for interactive design exploration, knowledge not found in existing tools or treatments on refactoring.

- An taxonomy of fifty-two refactorings, organized around thirteen refactoring verbs. Each refactoring verb is a generic refactoring action that can apply in many different contexts. With this taxonomy, the space of possible refactorings can be enumerated, and new refactorings identified.

Chapter 4

The Refactoring Knowledge Base

This chapter describes the content and structure of DR. JONES' knowledge base of refactorings. This chapter answers the question, *What does Dr. Jones know about refactoring?*

The content of DR. JONES' knowledge base reflects its focus as a design exploration assistant. As discussed in Chapter 3, source refactories are focused on the preconditions and syntax transformations of refactorings. DR. JONES takes a different view of refactoring, focusing on the knowledge needed to interact with the programmer as a redesign assistant. An assistant requires additional knowledge (beyond preconditions and source transformations) to provide feedback and guidance throughout the redesign process.

This chapter first discusses the overall structure and coverage of the knowledge base. It then presents a typical knowledge base entry in detail. The first refactoring from the Bicycle example, Specialize frameType, will be used to illustrate how a knowledge base entry is represented and used. Issues in the knowledge engineering of DR. JONES will also be discussed.

4.1 What Refactorings Dr. Jones Knows

Figure 4-1 lists the twenty-two refactorings implemented in DR. JONES' knowledge base. This is a subset of the fifty-two refactorings found in the entire refactoring space. These twenty-two refactorings were implemented because they are used in the Bicycle and JUnit design scenarios, which are discussed in Chapter 7.

These refactorings are presented with explanatory keywords, along with their verbs and arguments. This longer, sentence-like form is used to display the refactorings to the user in DR. JONES, because it helps explain their purposes and effects.

The knowledge base is structured as a set of semi-independent entries. Each entry is self-contained and corresponds to a single refactoring. Removing a refactoring does not directly disable other refactorings in the knowledge base. However, because a refactoring may require or suggest other refactorings, removing refactorings does degrade DR. JONES' ability to provide design guidance.

Create *in* aPackage *a class named* aClassName
Create *in* aClass *a method named* aMethodName
Create *in* aClass *a field named* aFieldName
Create *for* aMethod *a parameter named* aParamName
Remove aMethod
Rename aPackage *to* aPackageName
Rename aClass *to* aClassName
Rename aMethod *to* aMethodName
Rename aField *to* aFieldName
Move aMethod *from* oldClass *to* newClass
Hide aMethod *with* narrowerModifier
Reveal aMethod *with* widerModifier
Reveal aClass *with* widerModifier
Decompose aClass *into a class named* aClassName
Decompose aMethod *into a method named* aMethodName
Encapsulate aField *with accessor methods*
Encapsulate aTypeCodeField *to a class with subclasses* valueField_1 ... valueField_n
Generalize class_1 class_2 ... class_n *to a superclass named* aClassName
Pull Up method_1 method_2 ... method_n *to* aSuperClass
Specialize aClass *to a subclass named* aClassName
Specialize aTypeCodeField *to subclasses named* valueField_1 ... valueField_n
Push Down aMethod *to subclasses* subClass_1 subClass_2 ... subClass_n

Figure 4-1: The entries in the refactoring knowledge base.

Figure 4-2 illustrates the dependencies among the refactorings in the knowledge base. A refactoring R_1 depends on another refactoring R_2 if DR. JONES may require or suggest R_2 as the result of R_1. It is interesting to note that if one ignores self-edges, the graph in 4-2 is acyclic; this means that any dialogue beginning with a single refactoring command will be finite. (The self-edges in the graph arise from method overloading in Java; whenever the programmer moves, renames, pushes down, or pulls up a method, DR. JONES suggests that he do the same to its overloaded variants.)

The dependencies divide the refactorings roughly into three types. Complex refactorings, like Specialize aTypeCodeField, depend on many others. Intermediate refactorings, like Move Method, have both incoming and outgoing dependencies. The simplest refactorings, Hide and Reveal, depend on no others.

4.2 What Dr. Jones Knows About a Refactoring

DR. JONES has five kinds of knowledge about each refactoring in its knowledge base.

1. *Must-guards* are conditions that must be satisfied for DR. JONES to permit the refactoring. A must-guard can have a repair action, which is offered to the programmer if its condition is not satisfied initially. The repair action is intended to change the

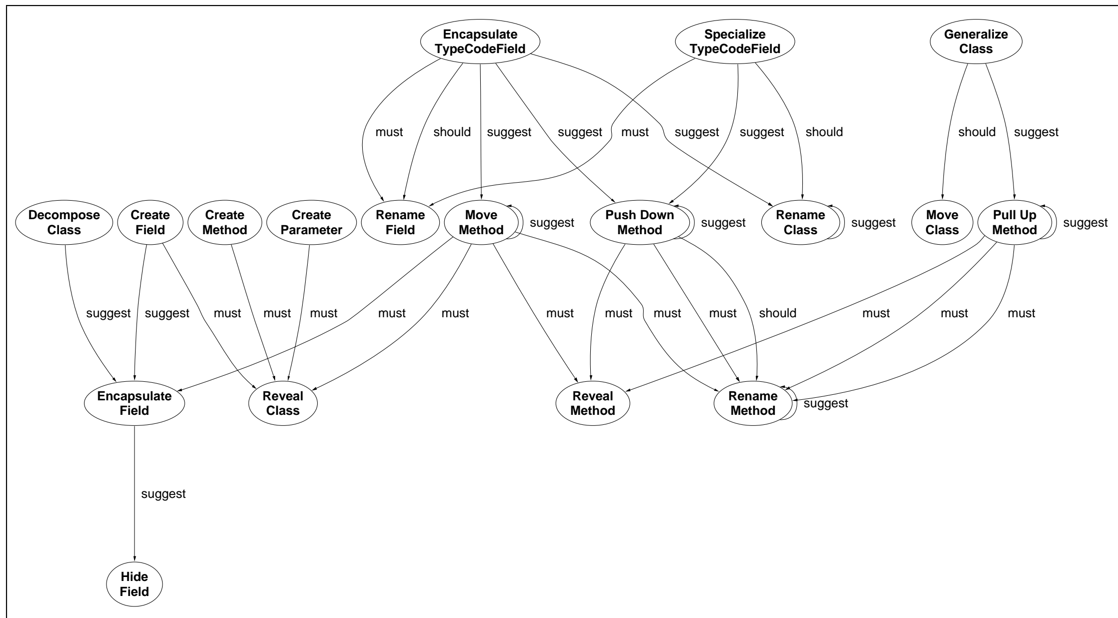

Figure 4-2: The graph of dependencies in the knowledge base.

design or the refactoring so that the must-guard will become satisfied.

2. *Should-guards* are conditions that prevent a refactoring from degrading the design in some way. If they are left unsatisfied, the programmer is warned, but he may choose to ignore the warning and do the refactoring anyway. Like must-guards, should-guards can have repair actions that fulfill unsatisfied conditions.

3. The *design transformation* that modifies DR. JONES' design representation to reflect the refactoring.

4. *Design suggestions* in the form of additional refactorings that can be done immediately after this refactoring.

5. *Editing directions* for the source code to execute the refactoring.

A sample knowledge base entry for the Specialize aTypeCodeField refactoring is shown in Figure 4-3. This is the first refactoring done in the Bicycle example, where it is used in this form:

Specialize frameType *to subclasses named* HYBRID MOUNTAIN ROAD

These entries are really specifications for the actual knowledge base, which is implemented in Java. Appendix B contains a complete set of entries like Figure 4-3 for DR. JONES' knowledge base. The remainder of this section explains this entry's notation and tours its contents.

Specialize aTypeField *to subclasses named* $\text{valueField}_1 \ \ldots \ \text{valueField}_n$

Must Guards	
M1	$\forall i$ NoNameConflict(NameOf(valueField_i), PackageOf(ClassOf(aTypeField))) $\hookleftarrow$ Rename valueField_i
M2	$\forall i$ aTypeField.$type$ = valueField_i.$type$
M3	IsFinal(aTypeField)
M4	$\forall i$ IsFinal(valueField_i)
M5	$\neg\exists method$ Uses($method$, aTypeField) $\wedge$ (ClassOf($method$) $\neq$ ClassOf(aTypeField))
M6	$\neg\exists subClass$ IsA($subClass$, ClassOf(aTypeField))

Should Guards	
S1	$\perp$ $\hookleftarrow$ Remind user to check that aTypeField only takes values from $\text{valueField}_1 \ \ldots \ \text{valueField}_n$
S2	$\forall i \ \forall package$ NoNameConflict(NameOf(valueField_i), $package$) $\hookleftarrow$ Rename valueField_i

Design Transforms	
X1	$\forall i \ C_i \leftarrow$ +Class(NameOf(valueField_i), PackageOf(ClassOf(aTypeField)))
X2	$\forall i$ +IsA(C_i, ClassOf(aTypeField))
X3	$\forall i$ $-$ valueField_i
X4	$-$aTypeField

Design Suggestions	
R1	$\forall method$ Uses($method$, aTypeField) $\rightarrow$ Push Down *method to subclasses* $C_1 \ \ldots \ C_n$
R2	$\forall i \ \neg$StandardClassName(C_i) $\rightarrow$ Rename C_i

Figure 4-3: The knowledge base entry for the Specialize aTypeCodeField refactoring.

4.2.1 Must-Guards

A must-guard represents a condition that must be satisfied for DR. JONES to permit the refactoring. If a must-guard for a refactoring fails, it indicates that the refactoring will create an invalid program, or the behavior of the program will likely change. Neither of these consequences are desirable.

Each must-guard consists of a condition and an optional repair. The condition is a boolean query that is tested against the current design of the program. The repair is an action recommended to the programmer when the condition is not satisfied. The action is intended to transform the design so that the condition will become satisfied.

The repair for a must-guard can be one of four kinds of actions:

- *Apply a prerequisite refactoring.* In many cases performing one refactoring enables another to succeed. For example, when moving a class C from one package to another, its visibility must be wide enough to ensure that existing references to it remain valid. First applying the Reveal C as public refactoring will ensure this is the case.

- *Modify the arguments to the refactoring.* Sometimes adjusting one or more of the refactoring's arguments will make it applicable. For example, one must rename a method in a way that avoids name conflicts with existing methods. When this is the case, adjusting the newName argument should remedy the conflict.

- *Abort.* Not all must-guards have repairs. If no repair is specified, DR. JONES aborts the attempted refactoring. The programmer may then attempt a different sequence of refactorings to achieve his redesign goals.

Figure 4-3 shows the must-guards, M1-M6, for the Specialize aTypeCodeField refactoring. Recall that the purpose of this refactoring is to replace a field being used as a type code, by creating subclasses for each of the type code values. The subclasses can then contain behaviors that used to depend on the type code.

In the specification, guard conditions are written in a form derived from first-order logic. The standard boolean expressions are used: $\top$, $\bot$, $\wedge$, $\vee$, and $\neg$, for true, false, and, or, and not, respectively. $\forall$ and $\exists$ represent universal and existential quantification over variables, i.e., the enclosed condition is true for all or some value of the variable. The scope of quantified variables is limited, in that they can only take values from DR. JONES' design representation, or an argument index $1 \ldots n$.

The notation is extended with functions and predicates. Functions return values, while predicates always return $\top$ or $\bot$. For example, the functions NameOf(), ClassOf(), PackageOf(), and TypeOf() return the name, class, package, and type values of their argument.

The predicates require a little more explanation:

- StandardClassName($name$) returns $\top$ if $name$ follows the conventions for naming classes in Java.

- NoNameConflict($name$, $package$) returns $\top$ if there is no other class named $name$ in $package$.
- IsFinal(elt) returns $\top$ if elt is marked as final (immutable) in the program.
- IsA($class_1$, $class_2$) returns $\top$ if $class_1$ directly extends or implements $class_2$.

These functions and predicates are evaluated directly on DR. JONES' design representation; no general inference capability is used to prove them. For example, to evaluate NoNameConflict($name$, $package$), DR. JONES invokes a method that compares $name$ to the names of all the classes in $package$, and returns true when no name matches.

Returning to Figure 4-3, guards M1 and M2 ensure that the names of the new subclasses are legal and won't cause current name conflicts. If there is such a conflict, DR. JONES suggests that the programmer rename the value field before proceeding. (The $\hookleftarrow$ can be read as "potentially satisfies.")

Guards M2, M3, M4, and S1 (discussed below) ensure that frameType is a true type code; that is, it takes exactly one of the fixed set of values HYBRID, MOUNTAIN, and ROAD over its lifetime. This is important, because if frameType takes values besides these, or changes its value over its lifetime, then this refactoring is not applicable; after the refactoring the type code field will be replaced by an object's membership in HYBRID, MOUNTAIN, or ROAD, and an object can never change its class in Java.

Guard M5 ensures that there isn't behavior outside of Bicycle that depends on frameType; such variations won't be captured in the new subclasses. Guard M6 ensures that there are no subclasses of Bicycle, hence the refactoring won't conflict with existing subclasses.

4.2.2 Should-Guards

Should-guards are similar to must-guards, except that DR. JONES still allows the refactoring even if they are unsatisfied. Should-guards are generally intended to prevent the refactoring from introduction a design flaw into the program, or to remind the programmer to check conditions that DR. JONES cannot.

Guard S1 is one of these latter cases; DR. JONES cannot check it because it lacks sufficient program information to do so. Implementing the check requires a dataflow analysis to find the values that could possibly be assigned to aTypeCodeField in class initializers or constructors (Opdyke, 1992). Instead, it remains unsatisfied, causing its repair to remind the programmer to verify the condition before implementing the refactoring. This is a tradeoff made by DR. JONES' working at the design level, instead of the source code level.

Because of conditions like S1, it is difficult to construct proofs (even in outline form) that show that a DR. JONES refactoring preserves behavior, if it were applied to the source. The guards are intended to guide design exploration, and not guarantee behavior preservation.

Specialize frameType has one other should-guard S2, which checks for name conflicts in all packages. This prevents a design flaw in which two classes with the same name (but in different packages) are both imported by a third class, requiring the ambiguous class names to be fully qualified wherever they are used.

4.2.3 Design Instance Transformation

The next part of the knowledge base entry describes how DR. JONES transforms the current design to reflect the refactoring. In the notation, $+$ indicates that new elements are added to the design, and $-$ indicates that elements are removed. The $C_i \leftarrow$ binds the newly created class to C_i, so that they be referred to elsewhere in the specification.

For Specialize frameType the design transformation steps are to

1. Create new classes C_1, C_2, and C_3 named HYBRID, MOUNTAIN, and ROAD.

2. Create new IsA(C_1, Bicycle), IsA(C_2, Bicycle), and IsA(C_3, Bicycle) dependencies.

3. Remove the frameType field from the Bicycle class.

4. Remove the three value fields ROAD, HYBRID, and MOUNTAIN from the Bicycle class.

4.2.4 Design Suggestions

R1 suggests that the programmer Push Down any method that uses frameType to the new subclasses. These methods, like checkSpecs(), are likely to have conditional behavior which depended on the type code. If the programmer follows the suggestion, and Push-es Down checkSpecs(), overriding methods are added to HYBRID, MOUNTAIN, and ROAD. Now, checkSpecs() can specialize its behavior according to the subclass of Bicycle.

This new ability to encapsulate behavior according to frame type is the payoff for performing this refactoring. Frame-specific behavior, which was hidden in the methods of Bicycle, now becomes explicit and localized in the new subclasses. There, it is easier to understand and maintain.

R2 checks if the names of the newly created subclasses follow naming conventions for Java classes, and suggests the programmer rename them if they are non-standard. In this case, DR. JONES suggests that the programmer Rename HYBRID *to* Hybrid, and so forth.

4.2.5 Source Editing Instructions

Although DR. JONES is not a source refactory, its design representation keeps references to the original source locations of elements and dependencies. With this information DR. JONES can tell the programmer the places in the source that need to be modified to execute a refactoring. (DR. JONES does not attempt to update this location information across refactorings, however.)

For Specialize frameType, the source editing instructions are to:

1. Create source files to define the new HYBRID, MOUNTAIN, and ROAD classes.
2. Declare the new classes to extend Bicycle.
3. Remove the declarations of the HYBRID, MOUNTAIN, and ROAD fields.
4. Remove the declaration of the frameType field.
5. Convert existing uses of frameType to instanceof expressions.

This part of the knowledge base is omitted from Figure 4-3 and Appendix B, as the source editing instructions are not directly available to the programmer through DR. JONES' user interface, and source transformation is not the focus of this research.

4.3 Knowledge Engineering of Dr. Jones

A substantial portion of the effort in this research involved prototyping and implementing DR. JONES' knowledge base of refactorings. Each refactoring went through several versions, starting with an analysis of the informal written description as found in *Refactoring* (if one was found there), to a semi-formal prototype entry, to an implemented class in DR. JONES, and ending with specifications like Figure 4-3. Each step in this process created refactoring descriptions of increasing refinement.

4.3.1 Knowledge Base Development

The knowledge base was developed in an incremental fashion. The first step was collecting available information about refactorings. Martin Fowler's *Refactoring* (Fowler, 1999) provided the source material for much of this effort. An analysis of his refactorings led to the formulation of the refactoring space of Table 3.2. The goal of knowledge base development then became to implement all of the refactorings in the refactoring space.

Once that goal was established, knowledge base entries for the refactorings in the space were prototyped in semi-formal English that included variables and logical connectives. Part of such a prototype is shown in Figure 4-4. Although the structure and terminology of entries has evolved from these prototypes (compare Figure 4-4 with Figure 4-3), their basic content has remained the same.

Once about fifty of the prototypes were done, Move Method was chosen as the first refactoring to implement. This refactoring was judged to be complex enough to expose implementation issues, because it both required and suggested other refactorings, and had complex guard conditions. It turned out to require the most lines of code of any refactoring (333).

```
rule RENAME-METHOD-GUARDS (aMethod aClass oldName newName)

;; requirements
newName is a legal java identifier / illegal identifier / let user fix name
oldName != newName / same name / let user fix name
aMethod is not a constructor / can't rename constructor / ABORT
no other method, field, or inner class in aClass, its superclasses,
   or its subclasses is named newName /
   name conflict / let user fix name

;; warnings
newName is a standard method name [a-z]+([A-Z][a-z]+)+ /
       unstandard name / let user fix name
a visible class, field, or parameter has the same name, THEN
       it's legal, but warn the user that the name is ambiguous
```

Figure 4-4: Part of a prototype entry for Rename Method in the Dr. Jones knowledge base.

After an initial version of Move Method was implemented in DR. JONES, the other refactorings were added one at a time as required to complete the Bicycle and JUnit redesign scenarios. Each successive refactoring became easier and easier to implement, as behavior used by multiple refactorings was factored out and available for reuse. For example, the NoNameConflict() predicate used in the Specialize aTypeCodeField entry is used in several other refactorings, and so became a part of Refactoring, the base class for all refactorings in the system. Consult Appendix C for the source code of the Specialize aTypeCodeField refactoring for an example of how this framework was used.

A code template for implementing new refactorings also evolved, part of which is shown in Figure 4-5. The template contains empty definitions for the methods that need to be overridden in Refactoring, the base class. Variable parts of the template are enclosed in angle brackets. The template minimized the amount of repetitive coding needed to add a refactoring to DR. JONES.

Eventually, adding a new refactoring to DR. JONES took just a few hours of work, starting from the code template and the prototype description. That includes the time to evaluate the prototype, code the refactoring in DR. JONES, and correct any ambiguities and errors found by exercising it on test cases.

The refactorings were then rewritten in the more formal specifications found in Appendix B. This was done by examining the source code of the implemented refactorings and translating it into the notation.

4.3.2 Adding New Refactorings

Twenty-nine of the fifty-two refactorings in the refactoring space have yet to be implemented in DR. JONES. Therefore, it is worthwhile to ask how much effort remains to implement them. Because they already have entry prototypes, the remaining tasks are to refine the prototype entries and implement them in DR. JONES' framework.

```
public class <NewRefactoring> extends Refactoring
{

  private <type-1> m_<arg-1>;
  private <type-2> m_<arg-2>;

  private <NewRefactoring>(<type-1> arg-1, <type-2> arg-2, ...)
  {
    super();
    m_<arg-1> = <arg-1>;
    m_<arg-2> = <arg-2>;
    ...
  }

  public String getName()
  {
    return "<NewRefactoring>";
  }

  protected void evalMust(final DInstance s)
  {
  }

  protected void evalShould(final DInstance s)
  {
  }

  protected void apply(final DInstance s, final DInstance t)
  {
  }

  protected void evalDesign(final DInstance t)
  {
  }

  protected void evalSource(final DInstance s)
  {
  }

  public String toString()
  {
    return getVerb()+" "+<Description>
  }

}// <NewRefactoring>
```

Figure 4-5: The class template for refactorings.

However, the refactoring knowledge and its implementation details are not yet cleanly separated in DR. JONES. The implementer must have fairly detailed knowledge of the design representation data structures and must understand how to evaluate complex queries over them. Although these APIs have stabilized after the development of twenty-two refactorings, they require occasional revision when a refactoring demands a new kind of query. For these reasons, implementing new refactorings currently remains the province of DR. JONES' author alone.

Clearly, this should not always be the case; a programmer should have a way to specify and add refactorings to DR. JONES, without delving into the framework implementing them. Towards this goal, I have adopted a philosophy of implement first, generalize later. Once a large subset of the refactoring space has been implemented – perhaps forty refactorings or more – the collection of reusable primitives like LegalIdentifier(), NoNameConflict(), and so on, can form the basis of a specification language for refactoring. This language will enable the programmer to write new refactorings in a form similar to Figure 4-3, which hides DR. JONES' implementation details. DR. JONES would then become an interpreter for this refactoring language, instead of a system with embedded refactorings.

A specification language would ease implementation, but would not automatically produce new refactoring knowledge. If a user does wish to add a new refactoring to DR. JONES, he must have thorough knowledge of the semantics of Java (to come up with must-guards and source edits) as well as intuitions about good and poor design practice (to come up with effective should-guards and design suggestions). DR. JONES captures the knowledge of expert programmers, and so requires experts to extend its knowledge effectively.

4.4 Composing Refactorings

The kinds of refactorings considered in this research are intended to be the simplest design moves that programmers would want to make. Because of this, there are times when he wishes to achieve a higher-level design goal, like "decouple these two classes," or apply a design pattern, like the Visitor (Gamma *et al.*, 1995). This might require the application of dozens of individual design moves.

Prior work has shown that it is possible to systematically break down higher level design moves, such as the application of design patterns, into individual refactorings (Tokuda and Batory, 2001; Cinne'ide, 2000). The ability to compose refactorings flexibly would thus be a desirable feature of a program redesign tool.

DR. JONES supports the a simple way to compose refactorings with *scripts*, which are sequences of refactoring commands. The refactoring commands are read from a file and played back as if they were input through the main user interface. Scripts were primarily used to debug and demonstrate DR. JONES during its development.

Because scripts lack parameters, they cannot be reused across designs. However, adding parameters, variables, and basic control flow would be a straightforward extension to the current script implementation. This would permit the development of a generic script to (for example) apply the Visitor pattern to a set of classes in the design.

An example of a script, used to demonstrate the Bicycle redesign scenario, is included in Appendix D.

4.5 Summary

DR. JONES has a knowledge base for the design exploration of Java programs, based on twenty-two refactorings. DR. JONES' design exploration knowledge, such as which refactorings suggest and require others, distinguishes it from source transformation tools, which contain only refactoring preconditions and source transformations. DR. JONES' knowledge base was first prototyped with a semi-formal description, then implemented as working code, and finally specified in a more formal way in this dissertation.

A goal is to permit other programmers to contribute their own refactorings to DR. JONES by writing specifications like those in this dissertation. This will become possible when DR. JONES' knowledge is more cleanly separated from its implementation. Even so, the knowledge base has evolved to the point where the author can add new refactorings with a few hours' effort, so that implementing all fifty-two refactorings in the refactoring space is possible in the near term.

Chapter 5

Dr. Jones' Design Representation

To assist the programmer in redesigning a program, DR. JONES must have sufficient information about the program. This chapter answers the question, *What does* DR. JONES *know about the program being redesigned?*

DR. JONES maintains a design representation of the program that consists of two levels of information. The first level contains information about a particular design, i.e., a single point in the program's design space. This level is captured in DR. JONES' design instance (DI) representation.

The second level of information relates the designs the programmer has explored with DR. JONES, i.e., the program's design space as a whole. This information is captured in DR. JONES' design space (DS) representation. The DS is composed of design instances and the refactorings that generated one design instance from another. The root design instance, from which all others are derived, is constructed by analyzing the original program. This is done by automated reverse engineering of the source and compiled forms of the program, minimizing the programmer's up-front effort.

I also show how the representation's semantics could be extended to capture other kinds of information, such as design rationale generated during the redesign process, and compare it to other program representations in terms of its granularity, simplicity, and ease of generation. DR. JONES' representation, while less granular than other representations, contains the information essential for design exploration. Its relative simplicity make it easier to reason about and manipulate.

5.1 The Design Instance Representation

The design instance representation contains information about a Java program's design at a single point in its design space. However, because a DI is more abstract than the source code from which it is derived, it is possible to have a design instance without its corresponding source code. This fact lets DR. JONES use a design instance as a foundation that gives him the ability to see and manipulate future designs without having to transform

source code.

The DI representation was formulated with the following properties in mind:

- It is simpler than source code, by virtue of its abstractness. It captures information needed for DR. JONES to guide redesign, but hides many of the details present in the full program text. Because the representation is simpler, it is easier to query and manipulate.

- It is self-contained. This means it is possible to cache a design constructed from program analysis, work with designs even when their source code is no longer available, and save entire design spaces for later use.

- A design instance can be built automatically by analyzing a Java program's source and compiled forms, without additional programmer effort.

- Its semantics can be extended. New elements and dependencies can be added to the representation without disrupting DR. JONES' existing functionality.

5.1.1 A Design Instance Is A Graph

A design instance is essentially a graph. Nodes in the graph are the major elements of the program: its packages, classes, methods, fields, and method parameters. Nodes are related by a containment hierarchy, which follows the structure of Java: packages contain classes, classes contain methods and fields, and methods contain method parameters.

Nodes are also related by eleven types of dependencies, represented using directed edges between nodes. These dependencies capture how the definition of a program element depends on other elements; for example, IsA(Road, Bicycle) indicates that the Road class depends on the Bicycle class by inheritance. Figure 5-1 summarizes the containment relationships and dependency relationships among DI nodes.

Every node has a name, which is unique within the scope defined by the containing node. (Package names are globally unique.) For example, field names are unique within a class, and parameter names are unique within a method. The fully qualified name for any node can be constructed by appending the names of its ancestors in the containment hierarchy; for example, a method named $methodName$, contained in class $className$, contained in turn by package $packageName$, would have the fully qualified name:

$$< packageName > . < className > . < methodName >$$

Both nodes and dependency edges have other properties associated with them. For example, every node has the content of the element's JavaDoc-style documentation, and the source file and line number where the element was declared. Other properties specific to certain types of nodes are summarized in Table 5.1.

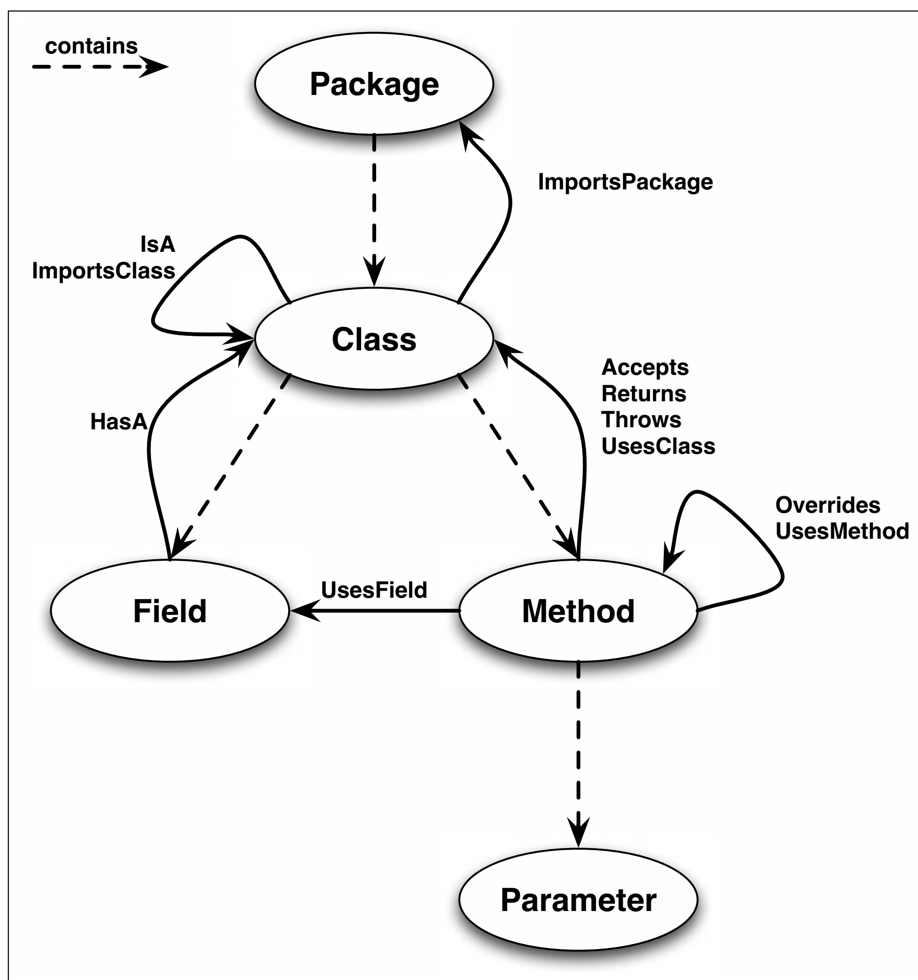

Figure 5-1: The containment and dependency relationships among nodes.

Node Type	**Property**	**Value**
Class	access	Access modifier, one of public or default
	isStatic	True if the class is static
	isAbstract	True if the class is abstract
	isInterface	True if the class is an interface
Method	access	Access modifier, one of public, protected, default, or private
	isStatic	True if the method is static
	isAbstract	True if the method is abstract
	isConstructor	True if the method is a constructor
Field	access	Access modifier, one of public, protected, default, or private
	isStatic	True if the field is static

Table 5.1: Node properties in the design instance representation.

5.1.2 Dependency Types

The design instance representation has eleven types of dependencies between program elements. They are Accepts, HasA, IsA, Overrides, Returns, Throws, ImportsClass, ImportsPackage, UsesClass, UsesMethod, and UsesField, and are summarized in Table 5.2. The HasA dependency has an additional type property, which indicates the multiplicity: exactly one (!), zero or one (+), or zero to many (∗).

Accepts($method$, $class$)	$method$ is declared to accept an object of class $class$
HasA($field$, $class^{[!,+,*]}$)	$field$ holds zero or more object(s) of class $class$
IsA($subclass$, $superclass$)	$subclass$ is a subclass of $superclass$, or $subclass$ implements the $superclass$ interface
Overrides($method_{sub}$, $method_{super}$)	$method_{sub}$ overrides $method_{super}$ by virtue of redefining it in a subclass
Returns($method$, $class$)	$method$ is declared to return an object (or array of objects) of type $class$
Throws($method$, $class$)	$method$ is declared to throw an exception of class $class$
ImportsClass($class_1$, $class_2$)	$class_1$ imports $class_2$ to bring it into its namespace
ImportsPackage($class$, $package$)	$class$ imports $package$ to bring all of the classes in $package$ into its namespace
UsesClass($method$, $class$)	an expression in the body of $method$ refers to $class$
UsesMethod($method_1$, $method_2$)	the body of $method_1$ contains an expression that invokes $method_2$ or one of its overriding methods
UsesField($method$, $field$)	the body of $method$ contains an expression that reads or writes $field$

Table 5.2: The dependency types in the design instance representation.

5.2 Building the Initial Design Instance

The initial design instance for a program is constructed by performing three analysis passes on its source and compiled forms; their results are merged to create the initial design instance.

5.2.1 JavaDoc Analysis

The first pass runs the JavaDoc documentation generation tool provided with the standard Java software development kit. Ordinarily it generates hypertext documentation for the program, but DR. JONES simply captures its underlying data structure and copies its contents to seed a new design instance. This gathers information on all the packages, classes, methods, fields, and parameters in the program, as well as their JavaDoc-style documentation. All of the information in the initial DI, except the Uses and HasA dependencies, is derived from this JavaDoc analysis.

5.2.2 Bytecode Analysis

The second pass generates the UsesClass, UsesMethod, and UsesField dependencies for the initial design instance. Because JavaDoc does not extract information from individual expressions in the program, such as which methods use which other methods and fields, a separate pass is needed. This is done with a bytecode-level analysis using the Byte Code Engineering Library (BCEL) [1]

DR. JONES uses BCEL to locate the codes corresponding to method invocations, field uses, and class references. For example, when the analysis finds a `NEW`, `INSTANCEOF`, `CHECKCAST`, `ANEWARRAY`, or `MULTIANEWARRAY` code corresponding to a class reference in the source, it generates a new UsesClass(M, C dependency from the method M containing the code to the referenced class C.

5.2.3 SuperWomble Analysis

In Java it is difficult to extract precise HasA dependencies in the presence of Java's generic collections. The Collection classes, like List, Set, and HashTable, aggregate values of type Object, the most generic class, and so give no indication of their actual contents. For example, if a Ranch class used a Set to represent a kennel of Dogs, without deeper analysis we could only conclude that a Ranch had one or more Objects (i.e., HasA(Ranch.kennel, Object*)). This relationship is too imprecise to be useful for redesign.

The SuperWomble tool uses dataflow analysis to discover the specific types of objects flowing into and out of such collections (Waingold and Jackson, 1999). For example, SuperWomble would be able to infer that HasA(Ranch.kennel, Dog*). DR. JONES invokes SuperWomble on the program and uses its results to add HasA dependencies to the program's initial DI.

5.2.4 Departures From The Java Specification

There are a some differences between the Java program structure as captured in DR. JONES' design representation, and the formal specification of Java (Joy *et al.*, 1998). These differences reflect choices that simplify the implementation of the representation and reduce the amount of reverse engineering effort involved.

- An interface, which is a class with only abstract method signatures and no implemented methods, is generally treated the same as a regular class. It retains its identity as an interface through its isInterface property. DR. JONES can use this property to prevent impossible situations, such as moving an implemented method to an interface.

[1] Bytecodes make up the compiled form of a Java program; they are interpreted by the Java virtual machine to execute the program.

- Inner classes (anonymous or otherwise) are not explicitly represented in the representation. Every Java program with inner classes can be rewritten to an equivalent program without inner classes, so this is not a fundamental restriction.

- Dependencies arising from the uses of fields and methods in static initializer expressions (for classes and fields) are not captured.

- Although packages form a containment hierarchy in Java, they do not in the DI, as these relationships are not captured by the JavaDoc analysis.

- Dependencies arising from the use of reflection (i.e., the dynamic lookup and use of classes, methods and fields) are not captured. Obtaining accurate dependencies in the presence of reflection may not be possible, and DR. JONES could warn the programmer this is the case when it finds reflection in the program.

5.3 The Design Space Representation

The second part of the design representation is the design space representation, which relates design instances. It is a directed tree of nodes, where each node corresponds to an entire design instance, and each edge corresponds to a refactoring that generated a new design instance from an existing one. Every design instance considered by the programmer, and the refactorings used to generate them, are included in the design space. The root of this tree is the initial design instance, constructed from the original program as described above. It branches when when the programmer revisits a design already generated, and refactors it in a different way.[2]

The design space representation also allows the programmer to annotate any design with a name, creating a bookmark in the space. He can also keep a To Do list with redesign tasks to perform, which are linked to each node in the space. Figure 5-2 shows a fragment of the design space created for the Bicycle example.

5.4 Comparing Program Representations

The design instance representation can be compared to other program representations with a similar purpose. Each representation has a certain granularity (in terms of how much it abstracts over the source), complexity, and ease of generation (from the programmer's perspective). Here, I compare abstract syntax trees, reflexion models (Murphy *et al.*, 2001), FAMIX (Tichelaar *et al.*, 2001), EMF (Griffin, 2002), and the Plan Calculus (Rich and Waters, 1987) to DR. JONES' design instance representation. DR. JONES strikes a balance among

[2]It is possible that the programmer arrives at the same design by two different sequences of refactorings; DR. JONES retains separate branches in the design space even if this is the case.

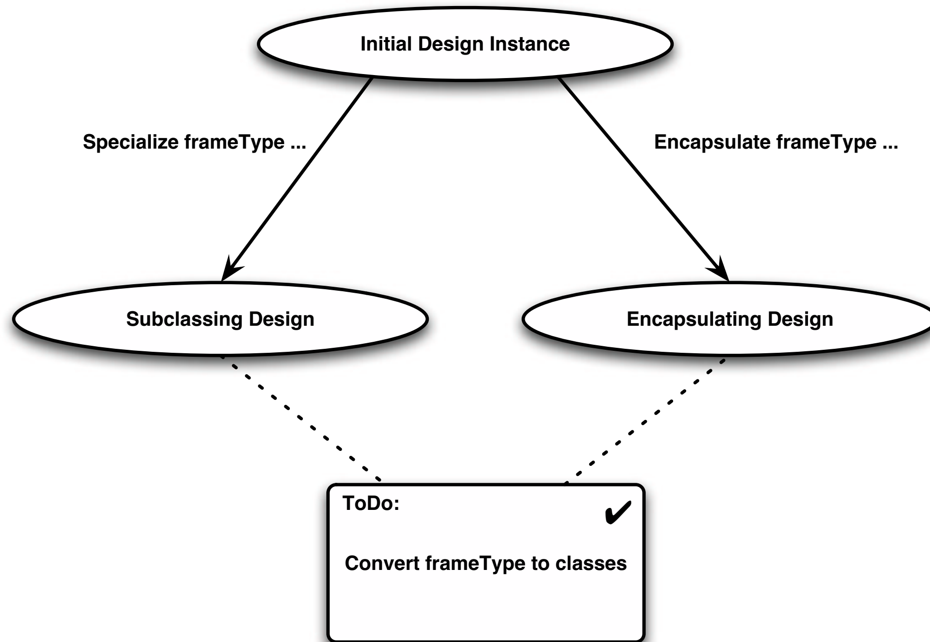

Figure 5-2: A fragment of the design space for the Bicycle example.

them by being simple (having only five node and eleven edge types), automatically generated, and of sufficient granularity for structural design exploration.

5.4.1 Abstract Syntax Trees

An abstract syntax tree (AST) is generated by parsing the source according to the Java grammar. This is a highly granular representation that contains nearly all the information in the source, and it can be generated automatically. However, additional analysis is required to extract the kinds of dependencies found in Table 5.2, because the uses of identifiers in the AST must be resolved back to their declarations. Making queries of the AST requires pattern-matching over the tree, which can be expensive. The representation is relatively complex, containing as many types as there are syntax and expression categories in Java.

5.4.2 FAMIX

FAMIX is a meta-model for language-independent refactoring (Tichelaar *et al.*, 2001). It is specified through an entity-relationship diagram, which contains a subset of DR. JONES' design instance representation: classes, methods, attributes (fields), InheritanceDefinitions (i.e., IsA), Invocations (UsesMethod), and Accesses (UsesField). The authors extend this language-independent core with additional information specific to Java. It has a similar granularity to the design instance representation, by omitting syntax-level and expression-level infor-

mation, but contains a subset of its content.

5.4.3 EMF

Like FAMIX, EMF is a meta-model. It represents the data model of a program domain. An EMF model includes the domain's classes, their operations, and the multiplicity and inheritance relationships among them. EMF is part of the Eclipse software development environment, and programmers can use Eclipse to generate the Java source code implementing an EMF model as well an editor for instances of the model. EMF models contain only the IsA and HasA dependencies from Table 5.2, and omit the other kinds of dependencies DR. JONES needs to provide redesign assistance.

5.4.4 Reflexion Models

Reflexion Models are used to assist programmers in comparing implementations and high-level models of programs (Murphy *et al.*, 2001). The reflexion model is not itself a program representation; rather it specifies a mapping between implementation fragments (like source files or functions) and high-level modules, which may not be explicitly represented in the source. Given this mapping, call graphs and other analyses can be presented in terms of the high-level modules, instead of the source fragments.

A reflexion model gives the programmer a way to make aspects of the higher-level program design explicit to programming tools, and generally requires little programmer effort to construct. Although a reflexion model is intended to enhance program understanding, as opposed to directly supporting design exploration, it could complement the design instance representation.

5.4.5 The Plan Calculus and CAKE

The Plan Calculus is the program representation in the Programmer's Apprentice (Rich and Waters, 1987). The Plan Calculus represents programs as graphs with nodes representing operations linked by edges representing data and control flow. This higher-level representation is reduced to propositional logic through the CAKE representation and reasoning system.

The Plan Calculus attempts to represent the program's behavior in a source-code independent form so that the Programmer's Assistant can reason about it. The Programmer's Apprentice required a library of clichés, which were program fragments hand-coded into the Plan Calculus. Reverse engineering programs of significant size into the plan calculus also proved difficult (Wills, 1992). In comparison, DR. JONES' design instance representation focuses on structural information in the program, and thus reasoning with it and reverse engineering into it are more tractable.

5.5 Capturing Design Rationale

A potential application of a design space representation is the capture of rationale behind redesign decisions. Capturing rationale has long been a goal of software engineering tools, but has proven difficult, because it ends up being more trouble than it is worth (at least to the programmer doing the redesign). Integrating rationale capture directly with the redesign process could help overcome this difficulty.

A typical ontology of design deliberation contains Issues, Positions, and Arguments (Conklin and Burgess-Yakemovic, 1996; Fischer *et al.*, 1996; Kunz and Rittel, 1970). In the domain of software, an Issue would be a design flaw or maintenance difficulty that motivates the redesign. A Position is a design alternative (obtained with DR. JONES) that resolves one or more Issues. An Argument is made to support or object to one or more Positions (designs). This deliberation embodies part of (re)design rationale, because it captures the motivations, alternatives, and arguments for and against the alternatives. Design rationale includes other kinds of information as well (Moran and Carroll, 1996), but this ontology serves as reasonable first approximation.

Figure 5-3 shows how the design space representation could be augmented to support this ontology of design deliberation. A sample deliberation for the two design alternatives in the Bicycle scenario is shown in Figure 5-4.

At the implementation level, the existing design space implementation could be extended by adding the appropriate classes. Because the new deliberation semantics are orthogonal to those of the existing design space, the new classes could be folded into DR. JONES without special difficulty.

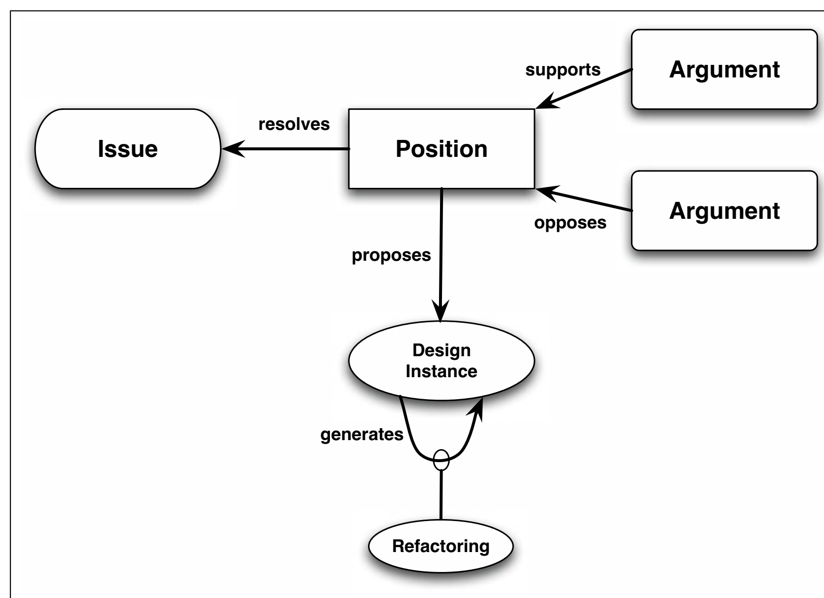

Figure 5-3: The design space representation extended to support design deliberation.

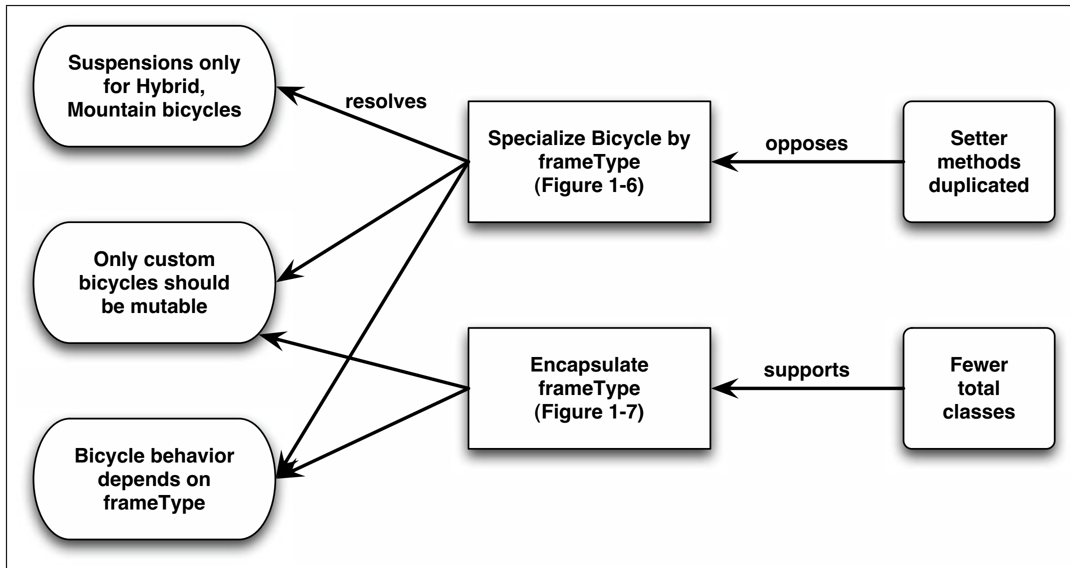

Figure 5-4: A deliberation that might have happened regarding the two designs in the Bicycle scenario.

5.6 Summary

DR. JONES represents its knowledge of the program with design instances, which capture individual program designs, and a design space, which relate design instances by refactorings. As compared to other program representations, they are simpler (containing only five node and eleven edge types) and less granular (omitting most information about expressions in the program). However, they contain enough information to support DR. JONES' design reasoning, and navigation of the design space by the programmer.

Chapter 6

Exploring Designs With Dr. Jones

This chapter answers the question, *How does the programmer interact with* DR. JONES*?* The user interface to DR. JONES addresses four challenges faced by a design exploration tool for software:

1. Visualize the program's design in a way that does not overwhelm the programmer. Diagrams with too much visual complexity confound sensemaking, which is critical to planning redesign.

2. Understand what the programmer intends when he gives a refactoring command to DR. JONES.

3. Manage the dialogue between the programmer and DR. JONES after a command has been given. This dialogue evolves through the design suggestions DR. JONES makes and the programmer's responses to them.

4. Support the navigation of the design space generated during a redesign session with DR. JONES.

Among these four challenges, DR. JONES meets the first and third with novel solutions that contribute to the design of user interfaces for software design exploration. The first challenge is met by the use of *multiple level-of-detail rendering* for software design diagrams. The the third challenge is met by a *dialogue tree* that presents and organizes the feedback DR. JONES provides. The other challenges are met with more conventional solutions.

I use the interaction between DR. JONES and the programmer for the first refactoring in the Bicycle example (outlined in Section 1.4) to illustrate these aspects of DR. JONES' user interface. Recall this refactoring specializes the Bicycle class by using the three possible values of its frameType type code field.

6.1 Visualizing The Current Design

Once the programmer specifies the locations of the source and compiled versions of the Bicycle class, DR. JONES renders it as shown in the screen shot in Figure 6-1. The diagram notation used by DR. JONES is a simplified variant of the UML class diagram (Booch *et al.*, 2001). Classes and interfaces are shown as rounded boxes, with arrows between them indicating their relationships (*is-a* and *has-a*). There is only one class visualized here, so there are no arrows. Abstract classes are distinguished from concrete ones by having a dashed border.

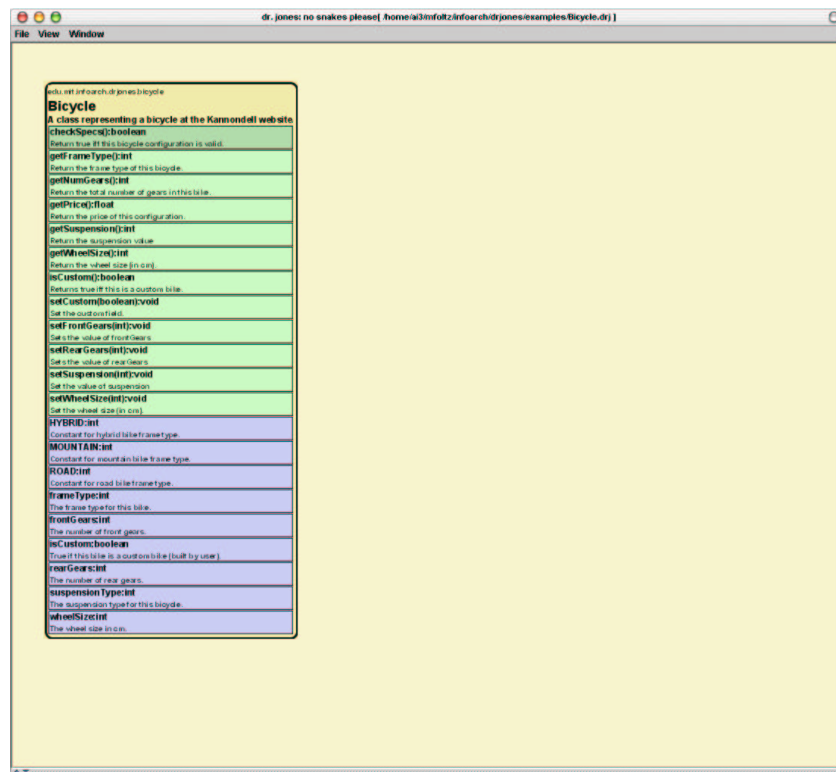

Figure 6-1: The initial rendering of the Bicycle class in Dr. Jones.

Each class contains a list of its declared methods and fields, with methods appearing above fields. Each member's declaration is followed by a short description, which is extracted from Javadoc comments in the source code. Methods and fields are rendered with different background colors to make them readily distinguishable.

Diagram layout is done with the GraphViz graph layout program (Gansner and North, 2000), which was modified to permit custom node objects and handle mouse events occurring within nodes.

6.1.1 Multiple Level-of-Detail Rendering

DR. JONES gives the user direct control over the visual complexity of its design diagrams. Without this control, software design diagrams can quickly become too information-dense, and thus unusable for sensemaking. Figure 6-2 (taken from a commercial development tool) shows this situation. In DR. JONES, the programmer controls diagram complexity by controlling the level of detail for each diagram element. Multiple-level-of-detail rendering has been used in other domains (Furnas, 1986) but has yet to be applied to software design diagrams.

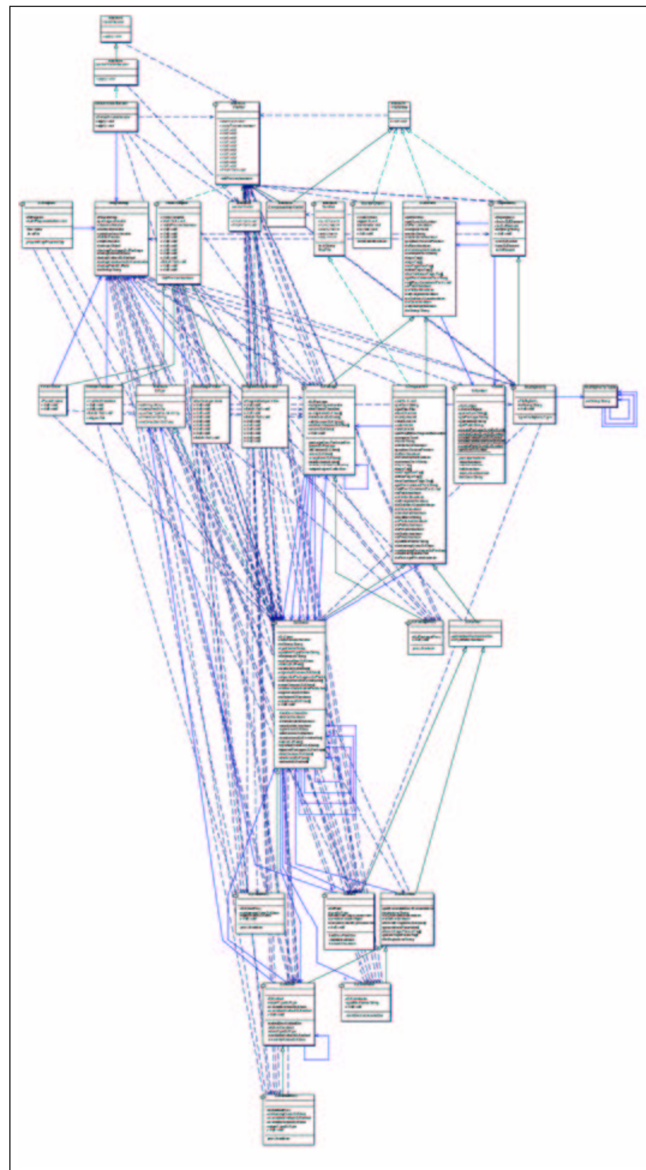

Figure 6-2: An overly complex design diagram produced by a commercial development tool.

Each element in the diagram, including classes, methods, fields, and relationships, has

ordered levels of detail defined for it. An element's minimum level of detail contains just enough information to identify it. An element's maximum level of detail contains all of its relevant information. Part of this range is illustrated for the Bicycle class in Figure 6-3.

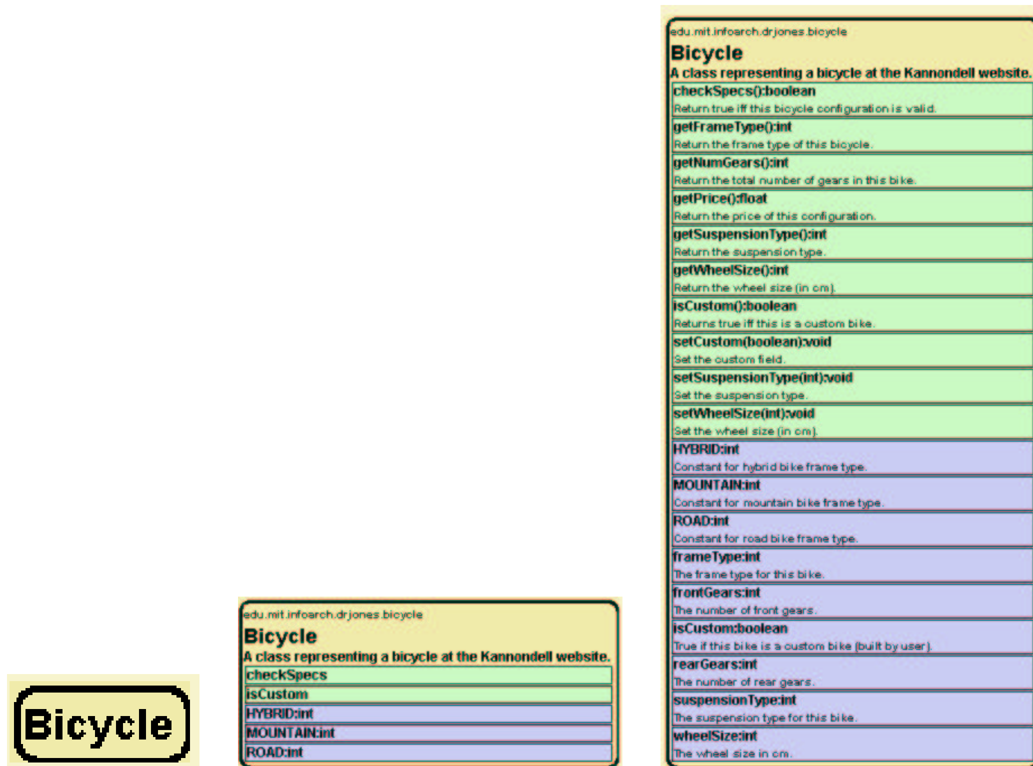

Figure 6-3: The minimum (left), default (middle), and maximum (right) levels of detail for a class.

Multiple-LOD rendering for classes is more complex than for other elements. Because a class' members carry a great deal of information about the class, the level of detail set for a class affects the levels of detail of its members. For example, increasing a class' LOD includes more of its methods and fields, and increases their respective levels of detail. Table 6.1 summarizes the six levels of detail available for rendering a class, and how they relate to the levels of detail for its contents as a whole. A check-mark indicates that the corresponding descriptive text is included for that level of detail.

Multiple level-of-detail rendering gives the programmer direct control over the visual complexity of design diagrams. In DR. JONES, the programmer can right-click on any element in the diagram to vary its level of detail, or to hide it entirely. This lets him match the information density and screen real estate of individual elements to his information needs. This approach differs from commercial tools, which typically offer only indirect control, such as setting filters on the diagram's contents.

[1] Excludes getter, setter, and delegate methods.

Contents	Class' Level of Detail					
	Min	2	3	4	5	Max
Class name	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
Class description		$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
Package name		$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
Public constructors			Min	Max	Max	Max
Public methods[1]			Min	Max	Max	Max
Public fields			Min	Max	Max	Max
All constructors					Min	Max
All methods					Min	Max
All fields					Min	Max

Table 6.1: The levels of detail available for rendering a class in DR. JONES.

6.1.2 Determining an Element's Level of Detail

If the programmer sets the level of detail for an element by clicking on it in the diagram, that LOD is always used. Otherwise a set of constraints are gathered to determine an element's LOD. An LOD constraint is an $(element, LOD)$ pair, which means that $element$ must be rendered at least at LOD LOD. The first source of constraints are the contextually defined LODs for methods and fields, given the LOD of their class (shown in Table 6.1). The other source of constraints is the output of a diagram consistency algorithm, described below.

Some elements in a software design diagram make no sense without appropriate context, like a method (which needs its enclosing class) or a dependency (which needs the elements it relates). To ensure that DR. JONES' diagrams make sense, it runs a consistency algorithm on the diagram's contents. The algorithm pulls in enough context for each element or relationship for it to be displayed meaningfully. In particular:

- Every dependency always includes the elements it relates.
- Every method or field always includes its enclosing class.

This consistency algorithm generates another set of $(element, LOD)$ constraints, which are combined with the contextually defined constraints to form a set C. The solution to C for each element e is simply

$$LOD(e) = max\ \{L|(e, L) \in C\}.$$

If no constraints are specified for e, it is omitted from the diagram. The diagram is then rendered using the levels of detail from the resulting solution C.

6.2 Interpreting the Programmer's Input

DR. JONES takes input from the programmer describing the intended refactoring, and matches it to a refactoring in DR. JONES' knowledge base. Returning to our example, Ecks wishes to specialize the Bicycle class by the frameType type code field. To tell DR. JONES to do so, he uses the diagram and the adjoining console user interface depicted in 6-4. The console consists of a row of buttons, one for each refactoring verb, and a display area for the dialogue concerning the refactoring in progress.

Ecks first clicks the Specialize button in the console, then on the elements in the diagram he wants to refactor. In this case, he clicks on the frameType field, and then the three value fields MOUNTAIN, HYBRID, and ROAD. As Ecks selects elements in the diagram, they are highlighted in red and displayed in the console. When he is done, he submits the command by clicking "Go."

DR. JONES takes the input vector

$$< \textsf{Specialize}, \textsf{frameType}, \textsf{MOUNTAIN}, \textsf{HYBRID}, \textsf{ROAD} >$$

and matches it to each refactoring in its knowledge base. The first refactoring whose verb and argument types match is selected, in this case

$$< \textsf{Specialize}, Field, Field^{+} > .$$

(The $Field^{+}$ means that one or more field arguments are matched.)

If no refactoring matches, the programmer is informed and he can try again. This straightforward approach works because the verbs divide the refactorings into groups in which ambiguous argument lists rarely arise.

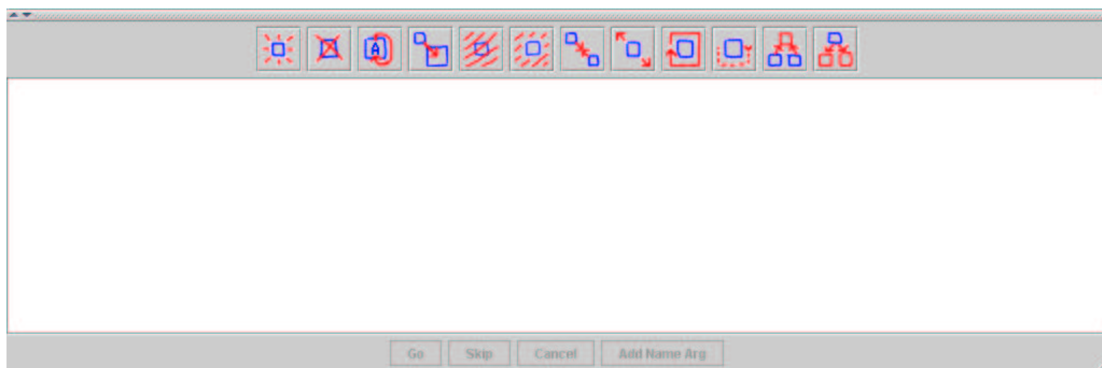

Figure 6-4: The command console for Dr. Jones.

6.3 Managing Dialogue

If an attempted refactoring succeeds immediately, and DR. JONES has no further suggestions to make, the console displays a check mark to indicate success and the programmer can start the next refactoring. Otherwise, DR. JONES engages the programmer in a dialogue, using the console to prompt him with suggestions based on the outcome of the initial refactoring command.

In Ecks' case, DR. JONES executes the Specialize refactoring immediately, and then makes some design suggestions based on its outcome. The first suggestion is to Push Down the checkSpecs() method. DR. JONES justifies every suggestion it makes with a short written explanation, in this case "because checkSpecs() made use of the frameType field." This suggestion is offered to Ecks in the console, where he can accept or decline it by clicking "Go" or "Skip." He can also back out of the whole refactoring dialogue at any time by pressing "Cancel," and revert to the design before the dialogue began.

Ecks accepts the suggestion to Push Down, but DR. JONES informs him he cannot do that right away. checkSpecs() is currently a private method, and it is not possible to push down a private method in Java. The Push Down refactoring is marked with an exclamation point, to indicate that it has pending prerequisites, and DR. JONES requires that Ecks first Reveal checkSpecs() to widen its visibility. Ecks accepts this requirement as well, and DR. JONES executes the Reveal refactoring, followed by the Push Down refactoring. The console resulting from this interaction is shown in Figure 6-5.

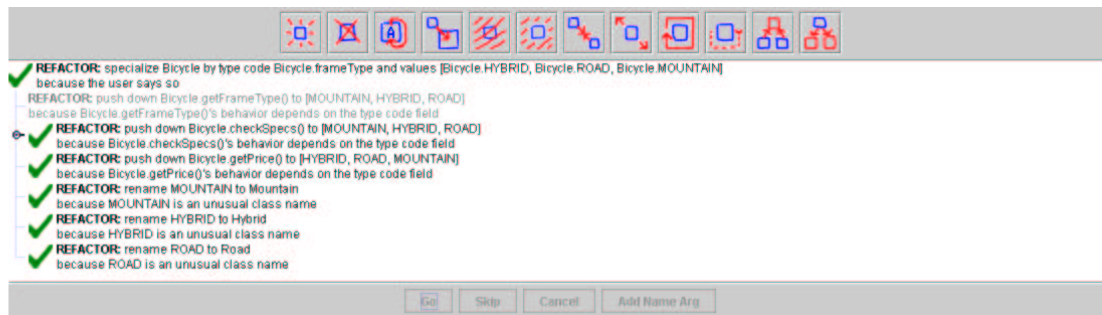

Figure 6-5: The console after the dialogue for the Specialize refactoring is complete.

6.3.1 The Dialogue Tree

As the example suggests, the interaction between DR. JONES and the programmer is modeled as a *dialogue tree*. The tree generated for the complete Specialize dialogue is shown in 6-6.

The root of this tree is a refactoring command the programmer has given to DR. JONES. A node's children are the actions DR. JONES requires or suggests based on the outcome of the parent node. An action can be a refactoring, a change to an argument in a pending

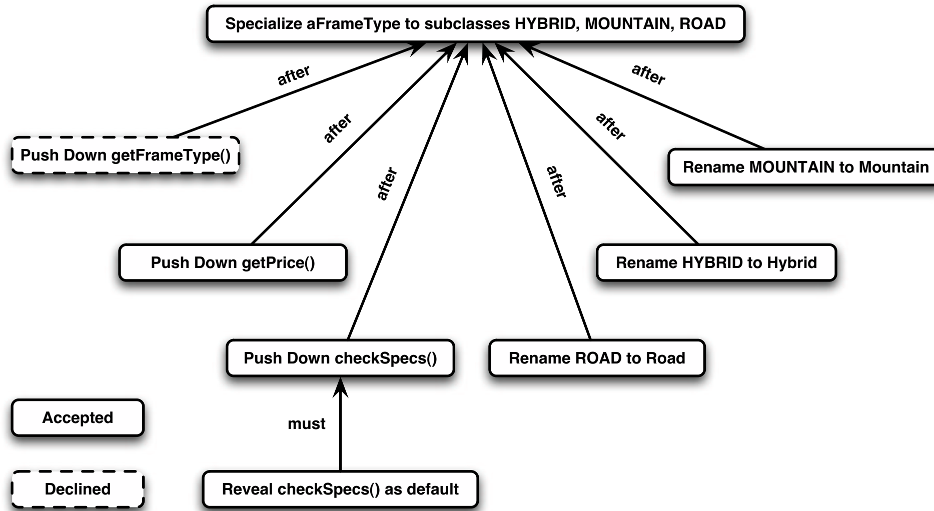

Figure 6-6: The dialogue tree generated for the complete Specialize dialogue.

refactoring, or a reminder (which is just displayed to the programmer).

There are three possible relationships between a child action and its parent:

1. A *must-child* is a prerequisite for its parent. If this action is declined by the programmer, or cannot be completed for any reason, the parent action is aborted.

2. A *should-child* suggests an action to be done before the parent action. The programmer can decline this action without affecting the success or failure of the parent action.

3. An *after-child* suggests an action to be done after the parent action is complete. The programmer can also decline this action without affecting the success or failure of the parent action.

These three types of children are created when Dr. Jones uses three corresponding parts of its knowledge for a refactoring. Must-children are created when a refactoring's must-guard is unsatisfied, and there is a repair action available to satisfy that guard. Should-children are created in the same way for unsatisfied should-guards, and after-children are created from the design suggestions Dr. Jones makes after a refactoring is completed.

The dialogue tree is built dynamically as the programmer accepts or declines Dr. Jones' suggestions. When the programmer accepts a suggestion, its children (if any) are generated and offered to the programmer. A parent's must-children are suggested first, followed by its should-children. Once all these children are complete, the parent action is executed, and finally its after-children are suggested to the programmer.

A dialogue tree resembles the goal-subgoal tree that might be produced by a planning algorithm, if one were were used to plan the execution of refactorings. However, dialogue

trees differ from goal-subgoal trees in two ways. First, no action on the design is taken without the programmer's consent, as any action could have undesired consequences that DR. JONES cannot predict. Requiring the programmer to confirm each action also keeps him fully informed about all the changes made to the design. Second, the dialogue tree contains many optional suggestions, which are not strictly required to meet refactoring goals. This differs from goal-subgoal trees, in which every child's goal must be satisfied to achieve the parent's goal.

6.4 Navigating The Design Space

As the programmer refactors designs, he traverses new points in the design space of the program. DR. JONES gives the programmer the ability to see the design space he has explored, and revisit prior designs to try design alternatives.

6.4.1 The Design Space Map

DR. JONES uses a map metaphor to present the design space, as shown in 6-7. The map is organized as a tree of nodes, each representing a design, which are linked by refactorings. The root of the tree is the initial design, and the tree grows downwards from the top of the map as refactorings generate new designs. By clicking on any design in the map, the programmer can instantly revisit it in DR. JONES' main window. If he chooses to refactor that design, a branch is created in the design space tree (as in Figure 6-7).

A checkbox allows the programmer to switch between viewing all designs in the space, or only the "top-level" designs that resulted from complete refactoring dialogues. This lets the programmer hide designs resulting from refactorings which were suggested by DR. JONES, and thus reduce the amount of screen space needed to render the map.

6.4.2 Bookmarks and the To Do List

When a new design is added to the design space, it is given a generated name of the form $*\mathsf{DNode} - \mathsf{nnn}*$, where nnn is a sequence of digits that guarantees uniqueness. Obviously, this does not communicate much about the content of the design, so DR. JONES gives the programmer the ability to rename any design as a bookmark, shown in Figure 6-8.

DR. JONES also supports using a To Do list to name designs. Since refactoring is often undertaken to remedy specific design flaws, DR. JONES lets the programmer keep a To Do list of problems and pending tasks. The programmer can add a To Do by providing a short written description of it. When the programmer completes a To Do, he checks it off the list, and the current design is bookmarked as "Fixed Todo n" (if no bookmark already existed for that design). Each design node keeps a record of the state of the entire To Do list, so that the programmer knows what remains to be accomplished when he revisits a prior design.

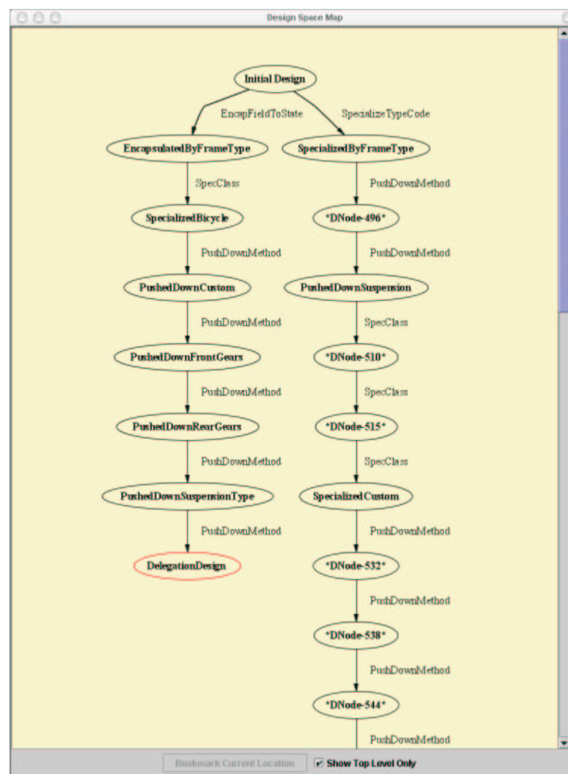

Figure 6-7: The Design Space Map lets the programmer see and navigate the design space explored during a redesign session.

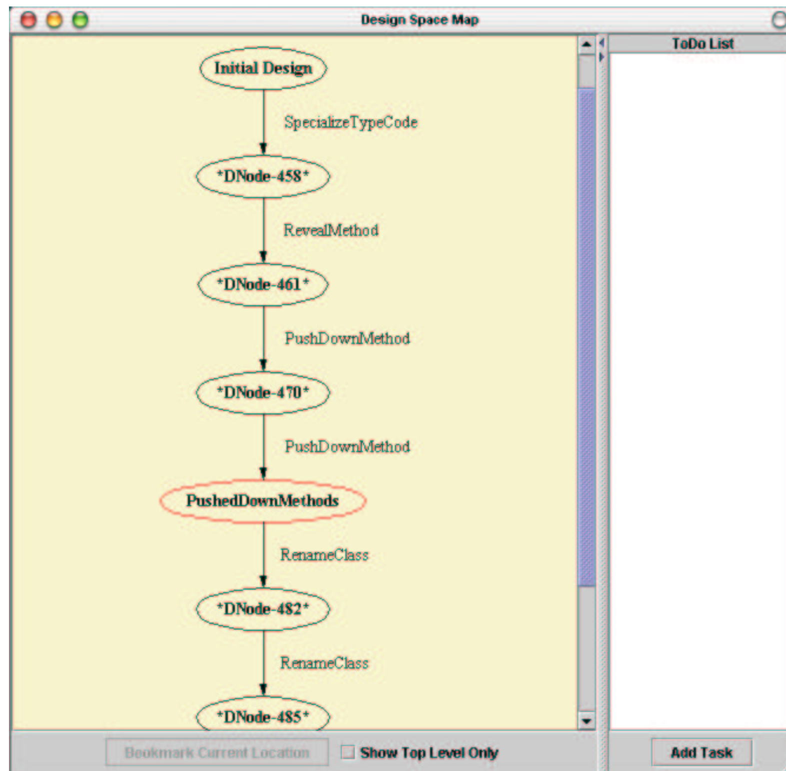

Figure 6-8: Bookmarking allows the programmer to create named designs.

6.5 Additional User Interface Capabilities

DR. JONES has additional user interface capabilities to assist the programmer in source browsing, and that were useful in the development and debugging of DR. JONES itself.

6.5.1 Inspecting The Design Space

The user can launch a Design Space Browser from a menu selection. This browser lets the user inspect the design space representation maintained by DR. JONES as a collapsible tree (Figure 6-9). The programmer can inspect the elements and relationships of all design instances in the design space. By double-clicking an element in the browser, the user can add it to the current design diagram. This browser was very helpful while debugging DR. JONES.

6.5.2 Source Inspection

At any time, the programmer can inspect the source of any program element shown in the design diagram by middle-button clicking. The corresponding source file is loaded into an Emacs buffer, with the cursor positioned at the beginning of the element.

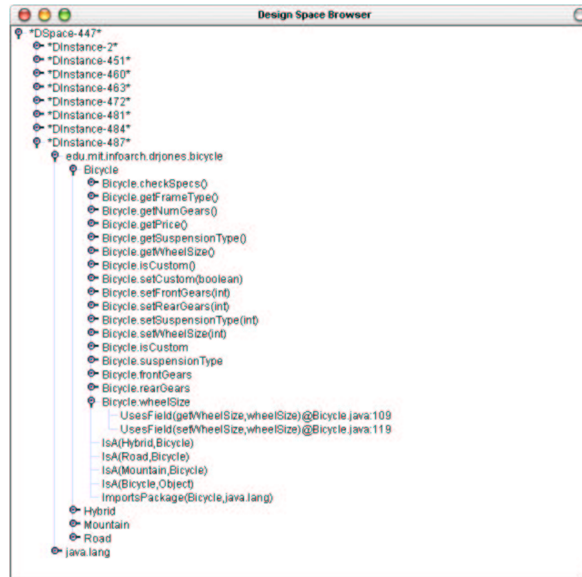

Figure 6-9: The Design Space Browser

6.6 Current Limitations of the User Interface

Diagram Rendering. Currently, DR. JONES gives the programmer direct control over the contents of design diagrams and the levels of detail at which they are rendered. Ideally, DR. JONES should be able to infer the contents of a good design diagram, and free the programmer from having to determine the diagram's contents at all. Also, the programmer should be able to get an overview by reducing the LOD of the diagram as a whole, and then drill down to expand details in a region of interest.

Input Interpretation. As more refactorings are added to DR. JONES, ambiguous inputs become more likely. And, we want to allow the programmer to express a refactoring command in multiple ways. A more general approach that parses inputs, instead of doing simple matching, would address these limitations.

Dialogue Management. Currently, the programmer must accept or decline every action suggested by DR. JONES. This becomes tedious for relatively safe refactorings like Reveal (widening visibility). Giving the programmer a way to automatically accept some suggestions would streamline interaction.

Design Space Navigation. In the two redesign scenarios described in this thesis, many of the designs in the design space were uninteresting because they were intermediate steps toward a larger goal, i.e. the application of a design pattern. Including these intermediate steps made the design space map larger than it needed to be. Detecting these larger moves, or encouraging the programmer to bookmark designs frequently, would help filter out irrelevant designs from the map.

6.7 Summary

To support design exploration, the user interface to DR. JONES must let the programmer see the current design, express his design intentions, receive feedback from DR. JONES, and navigate the resulting design space. Each of these activities brings with it a user interface design challenge. DR. JONES addresses these challenges and makes two contributions to to work on software design exploration tools. The first is multiple level-of-detail rendering of class diagrams that gives the programmer control over the diagrams' visual complexity. The second is a flexible dialogue management system that organizes and presents the suggestions DR. JONES makes to the programmer.

Chapter 7

Two Design Exploration Scenarios

DR. JONES is demonstrated here in two redesign scenarios. The scenarios were developed to illustrate how DR. JONES assists the programmer in design exploration, and to discover DR. JONES' strengths and weaknesses. Several lessons were learned in developing these scenarios, including both the strengths and limitations of DR. JONES' current approach, as well as opportunities for future work in this area. Each scenario involves multiple design alternatives, with each alternative the result of multiple refactorings of a starting design. As many as thirty-one refactorings were performed to generate each alternative.

The first scenario is the Bicycle redesign, which was developed as a case study by the author. A monolithic Bicycle class that combines multiple concerns is decomposed into multiple classes, encapsulating those concerns.

The second scenario evolves the design of JUnit, a unit testing framework for Java. JUnit is an open-source software project originally written by Kent Beck and Erich Gamma (Beck and Gamma, 2000). It has been in development for three years and has a large, established user base. When designing JUnit, its authors evolved a general unit testing framework from a simpler one by the successive application of design patterns (Gamma and Beck, 2000). I emulate this design evolution with DR. JONES, and also explore a design variation that support other kinds of tests (besides unit tests).

7.1 Decomposing Bicycle

The Bicycle class, shown in Figure 7-1, is part of a program that helps a bicycle manufacturer configure the bicycles it sells. It has multiple responsibilities in this initial form, and the programmer wishes to decompose this class to separate and encapsulate these responsibilities.

One responsibility of Bicycle is that it represents three types of frames, according to the value of its integer frameType field. This field can be one of the three constants HYBRID, MOUNTAIN, or ROAD, according to the three major frame types the manufacturer sells. Other methods in Bicycle, like getPrice() and checkSpecs(), behave differently according to

the frame type.

Another responsibility of Bicycle is that it represents bicycles that have suspension, i.e. shock absorbers for the front or rear wheels. However, suspension is not compatible with road frames, so the suspension responsibility and the frame type responsibility conflict; getSuspension() and setSuspension() aren't valid operations on a road bicycle.

The third responsibility is that the class represents both pre-configured bicycle models and custom configurations demanded by individual customers. Pre-configured bicycles should have an immutable configuration, so the Bicycle class should not provide setter methods for these bicycles.

These concerns are intermingled in the current design, and the programmer's task is to tease them apart with DR. JONES. The two alternatives considered here address the frame type concern to start, then address the other concerns with subsequent refactorings. Appendix D contains the source code of the Bicycle class, as well as a script of all the refactorings used.

7.1.1 Decomposition By Subclassing

The first design alternative uses subclassing to decompose Bicycle by frame type. The programmer tells DR. JONES to

Specialize frameType *to subclasses* HYBRID MOUNTAIN ROAD.

DR. JONES does so and suggests the programmer Push Down the getFrameType(), getPrice(), and checkSpecs() methods. The programmer accepts the suggestions for getPrice() and checkSpecs(), but does not push down getFrameType(), which just returns the (now unnecessary) type code value. checkSpecs(), being private, requires the Reveal checkSpecs() refactoring to be performed before it can be pushed down. DR. JONES also suggests renaming the new HYBRID, MOUNTAIN, and ROAD subclasses to Hybrid, Mountain, and Road. This dialogue results in the design of Figure 7-2.

The programmer now wants to isolate the getSuspension() and setSuspension() methods into the new Hybrid and Mountain subclasses. He uses Push Down to do so, and removes the base class methods after the Push Down is complete. He can do this because Bicycle has no clients; if there were existing invocations of the base class methods, DR. JONES would prevent him from removing them. (I return to this point later, in Lessons Learned.)

Finally, the programmer wants to isolate the setter methods in Bicycle. To do so he subclasses Hybrid, Mountain, and Road to CustomHybrid, CustomMountain, and CustomRoad, and then uses Push Down on the relevant setter methods in Bicycle. Again, he removes them from superclasses after he pushes them down, leaving them in only the Custom subclasses.

The final step is to recognize that the Custom subclasses now share a common set of setter methods. This represents unfortunate but necessary code duplication, because Java

edu.mit.infoarch.drjones.bicycle

Bicycle

A class representing a bicycle at the Kannondell website.

checkSpecs():boolean
Return true iff this bicycle configuration is valid.

getFrameType():int
Return the frame type of this bicycle.

getNumGears():int
Return the total number of gears in this bike.

getPrice():float
Return the price of this configuration.

getSuspension():int
Return the suspension value

getWheelSize():int
Return the wheel size (in cm).

isCustom():boolean
Returns true iff this is a custom bike.

setCustom(boolean):void
Set the custom field.

setFrontGears(int):void
Sets the value of frontGears

setRearGears(int):void
Sets the value of rearGears

setSuspension(int):void
Set the value of suspension

setWheelSize(int):void
Set the wheel size (in cm).

HYBRID:int
Constant for hybrid bike frame type.

MOUNTAIN:int
Constant for mountain bike frame type.

ROAD:int
Constant for road bike frame type.

frameType:int
The frame type for this bike.

frontGears:int
The number of front gears.

isCustom:boolean
True if this bike is a custom bike (built by user).

rearGears:int
The number of rear gears.

suspensionType:int
The suspension type for this bicycle.

wheelSize:int
The wheel size in cm.

Figure 7-1: The initial design of the Bicycle class.

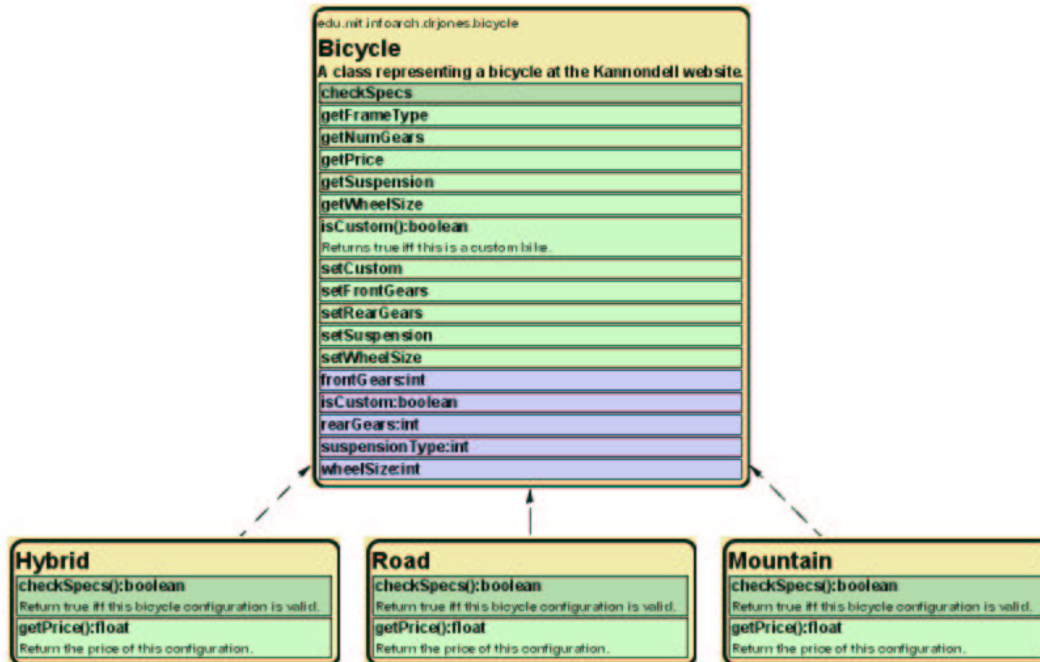

Figure 7-2: Bicycle decomposed into subclasses by frame type.

cannot mix in the setter implementations from a separate class. The best the programmer can do is to capture their common interface, by performing

Generalize CustomHybrid CustomMountain CustomRoad *to* "Customizable"

to create a new class Customizable (which DR. JONES creates as an interface by default.) DR. JONES identifies the common setter methods and suggests the programmer pull them up. This leaves the design in Figure 7-3.

7.1.2 Decomposition By Delegation

Another way to approach the separation of concerns is to encapsulate the frame type field into its own class, so that each Bicycle instance will have its own FrameType instance. Then FrameType can be subclassed to hold behavior specific to the hybrid, mountain, and road frame types. This behavior, which used to live in Bicycle, can then be delegated to the Bicycle's FrameType instance.

This approach is accomplished by the programmer applying the

Encapsulate frameType *to a class with subclasses* HYBRID MOUNTAIN ROAD

refactoring to the initial design. DR. JONES then takes the frameType-dependent methods and suggests moving them to the new FrameType class, and then pushing them down to

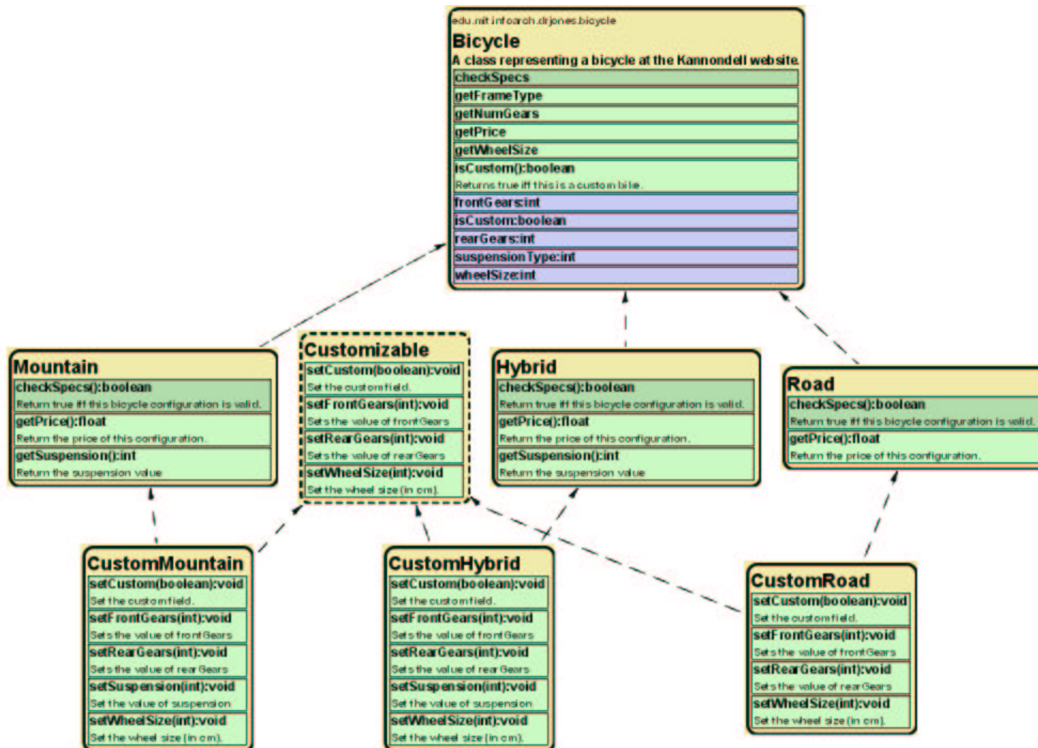

Figure 7-3: The subclassing redesign alternative for Bicycle.

its subclasses. The resulting design is shown in Figure 7-4. (This design move applies the State design pattern to Bicycle (Gamma *et al.*, 1995).)

The programmer can then make the distinction between customized and pre-configured bicycles by subclassing Bicycle itself. The programmer does so, creating CustomBicycle, and then pushes down the setter methods from Bicycle into it, resulting in the design in Figure 7-5.

Once this structure is in place, the suspension concern presents something of a dilemma. If the programmer moves the Bicycle.getSuspension() method to FrameType, pushes it down to Hybrid and Mountain, and removes it everywhere else, it will be available only for the frames that can have suspension, which seems correct. But, the programmer may still want to ask a Bicycle about its suspension without requesting its frame and checking the frame type. Also, the setSuspension() method is now in CustomBicycle, which would have to check the class of Bicycle.frameType and downcast it appropriately before actually changing the suspension type. These issues suggest that there's some room left to explore in this design; the programmer could proceed by further subclassing, or using a pattern like Null Object to represent a non-existent suspension.

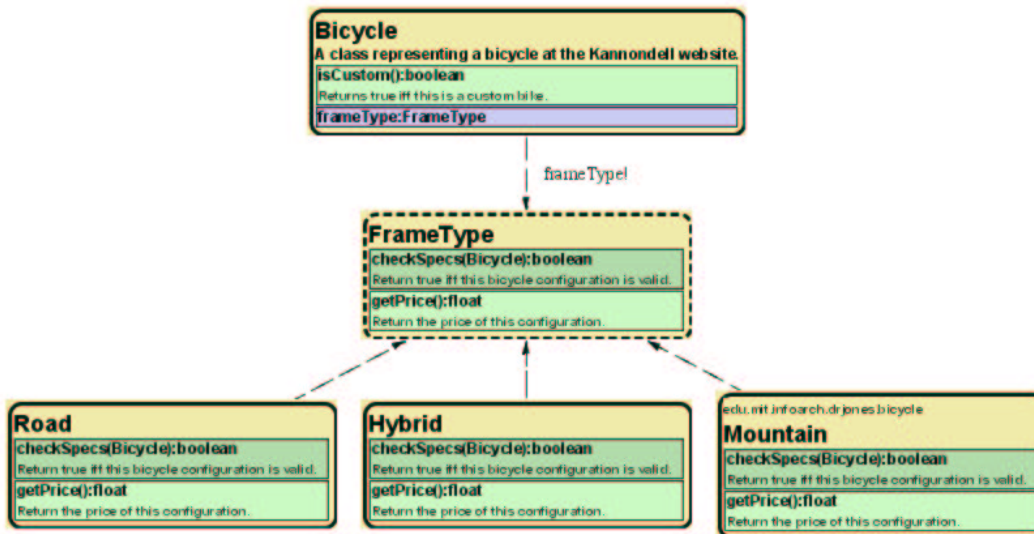

Figure 7-4: frameType encapsulated into its own class hierarchy.

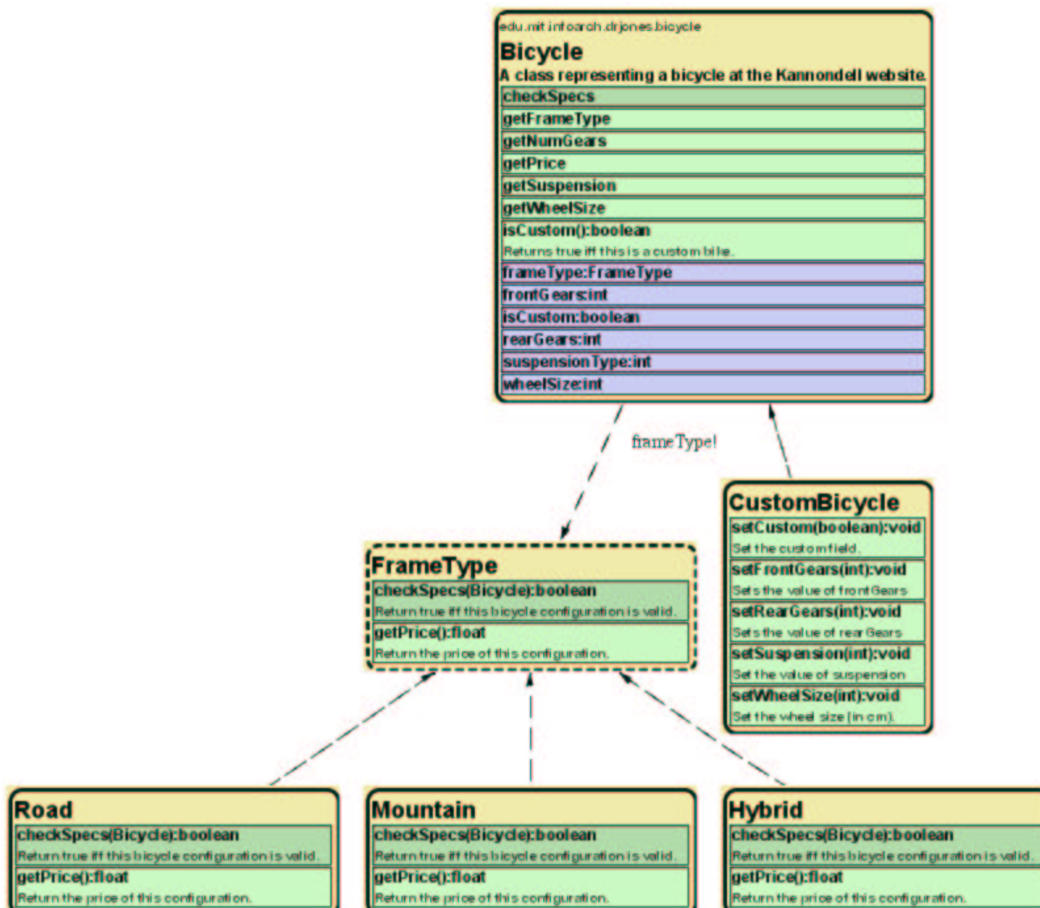

Figure 7-5: The delegation design alternative for Bicycle.

7.1.3 Lessons Learned

Several lessons were learned from this redesign scenario, illustrating both strengths and limitations of DR. JONES.

- **DR. JONES' strength is its simplicity.**

  DR. JONES was able to apply as many as twenty-seven refactorings to the design, without having to show the programmer a line of source code. DR. JONES focuses the programmer the program structure and the redesign task, instead of syntax and source formatting. DR. JONES also demonstrates simplicity in its interface, by having a focused task. An integrated development environment, which tries to support many tasks, has a much more complicated interface.

- **DR. JONES' assistance is helpful.**

  In nearly all cases, DR. JONES makes suggestions that emulate what a good programmer would do. For the Specialize frameType refactoring, the suggestions save the programmer the effort of going back to inspect the source code to find which methods to push down. This kind of assistance keeps the programmer focused on the redesign task.

- **DR. JONES doesn't know quite enough about the program.**

  Even though most of DR. JONES' recommendations do lead to better designs, it occasionally makes mistakes. In the first alternative, DR. JONES recommends that the programmer push down the Bicycle.getFrameType() method, even though it is no longer needed. This is because DR. JONES recommends that *every* method that uses frameType be pushed down. If DR. JONES knew that getFrameType() was just the getter method, it could instead suggest the programmer remove it.

- **The programmer can't see or manipulate the program's behavior.**

  One of DR. JONES' strengths it that it focuses the programmer on the structure of the program, leaving source level details behind. But refactoring is motivated by both structure and behavior, and it would be useful for the programmer to see and work with both. For example, the programmer plans to decompose Bicycle because he observes the frame-specific behavior in the original source code. And because DR. JONES does not update the source as he refactors, he cannot return to it to plan his next steps.

- **DR. JONES' representations become imprecise, and that limits the programmer's refactoring options in the future.**

  When DR. JONES performs certain refactorings that decompose behavior - like Push Down Method and Decompose Method - it makes the conservative assumption that all

the behavior in the original method is duplicated in its derivatives. Since DR. JONES' only representation of behavior is a collection of Uses... dependencies originating from a method, it copies all these dependencies. Without asking the programmer exactly how the expressions in the method are divided, or being able to infer the division itself, DR. JONES won't know how to distribute the Uses... dependencies accurately among the methods.

This means that, over time, UsesMethod, UsesField, and UsesClass dependencies will accumulate in the design representation that don't correspond to how the source would actually be refactored. For example, suppose the checkSpecs() method in Bicycle were

```
private boolean checkSpecs()
{
  switch (frameType) {
  case MOUNTAIN:
    return (getWheelSize() < 18 && getSuspension() == 2);
  case ROAD:
    return (getWheelSize() > 20);
  case HYBRID:
    return (getWheelSize() < 20 && getSuspension() == 1);
  default:
    // never happens
    return false;
  }
}
```

After the programmer pushes down checkSpecs(), the uses of getSuspension() will remain in Bicycle, although it would be reasonable to assume those expressions would be moved into the Mountain and Hybrid subclasses. These inaccurate uses later prevent the programmer from pushing down getSuspension() to the Mountain and Hybrid subclasses, although he should be able to do so.

- **Some conditions cannot be checked automatically.**

  As mentioned before, DR. JONES cannot conclude that frameType is a true type code, because it does not know that only the HYBRID, MOUNTAIN, and ROAD values are assigned to it. Instead it reminds the programmer that it cannot check this condition, and trusts him to remember (or verify) the status of frameType as a type code.

- **More flexibility in expressing refactoring intentions is needed.**

  It would be helpful to let the programmer express slight variations in his refactoring commands. One such variant for Push Down removes the superclass method after it has been copied to subclasses. This variant is used in the script for the Bicycle scenario by adding the `:remove` keyword to the Push Down commands. Allowing the

programmer to express variations like this through the user interface as well would help the programmer communicate his intentions to DR. JONES more accurately.

Several of these lessons suggest that better modeling of behavior would enhance DR. JONES' approach to design exploration.

7.2 Evolving JUnit

JUnit is a unit testing framework for Java. The programmer writes unit (i.e., class- and method-level) tests for his own program by extending classes in the JUnit framework and overriding their abstract methods. The framework then takes care of running the tests and reporting their results to the programmer. The framework also has a library of assertion methods which make it convenient to assert object equality, non-null references, and so forth.

7.2.1 Generalizing the Simple Framework

The authors of JUnit set out to use design patterns to evolve a very simple kernel framework (shown in 7-6) into a more general framework. Each successive pattern adds another kind of generality to the design. DR. JONES was used to emulate this design evolution by breaking the pattern applications into a sequence of thirty-one refactorings of the kernel framework. For the source code of the TestCase class, as well as a script of all the refactorings used, consult Appendix E.

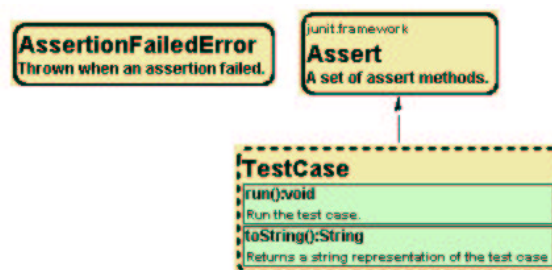

Figure 7-6: The kernel framework for JUnit.

The patterns applied to the framework were:

1. **Template Method.** This breaks the single run() method into setUp(), runTest(), and tearDown() methods. This is so test code in runTest() can share a common fixture of objects, created by setUp() and destroyed by tearDown().

2. **Collecting Parameter.** This creates a TestResult class to collect the results of a round of tests. A parameter of this type is passed to every run() invocation, and catches exceptions and assertion failures that occur during tests.

3. **Pluggable Selector.** This pattern lets the programmer choose a test to run by setting a name field in TestCase. The name field is used to look up and execute the corresponding test method.

4. **Composite.** This powerful pattern first creates a TestSuite class, which is a collection of tests, and then generalizes TestSuite and TestCase into a Test interface. The TestSuite can then hold a collection of Tests, which can be TestCases or other TestSuites. In this way the programmer can organize his unit tests into a tree, following the hierarchal organization of his program.

5. **Adapter.** The Adapter pattern is an alternative to the Pluggable Selector for choosing a test from a class of tests. The programmer creates an anonymous inner class, which overrides runTest() to invoke one of the programmer's own test methods. This is not a pattern so much as a strategy for overcoming Java's lack of method pointers.

6. **Observer.** The Observer pattern is used to keep the programmer informed of the progress of his test suite. In this case, the observed model is TestResult. The TestListener interface is extracted from TestResult, and TestResult is extended to hold a collection of these listeners. User interface components can them implement TestListener and register themselves with a TestResult to receive notifications of test successes and failures.

The design resulting from these patterns is shown in Figure 7-7.

7.2.2 Adding Test Variants

With DR. JONES I first evolved JUnit's design in the same way as its authors to create a more general framework for unit testing. As a design exploration experiment, I also formulated an alternative design with DR. JONES that extends the framework to accommodate some other kinds of tests:

- *Stress tests*, which spawn multiple threads that run a unit test simultaneously and repeatedly. These tests expose race conditions and other phenomena that arise only in those circumstances.

- *Pattern tests*, which are cookie-cutter unit tests that verify the implementation of design patterns like Singleton and Observer.

- *Regression tests*, which compare two sets of unit tests, and report those that succeeded in an earlier version of the program, but fail in the current version.

To develop this design alternative, I branched off the design space at the point after the Composite pattern had been applied. Each of these test variants is a subclass of TestCase,

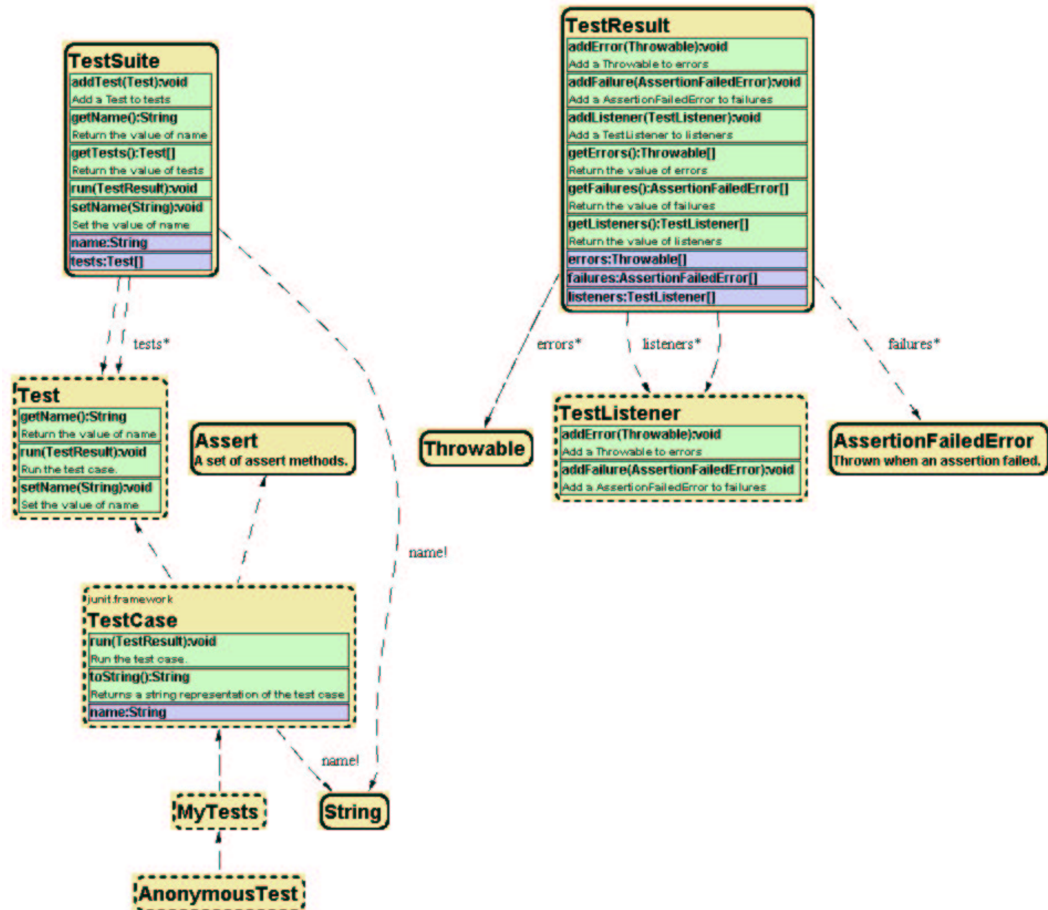

Figure 7-7: The evolved design of the JUnit framework.

so that it can be run like any other test in JUnit. However, each variant interacts with the rest of the framework in different ways. These variants are developed in the design in Figure 7-8.

The corresponding design space map for this scenario is quite large, and so is only shown schematically in Figure 7-9.

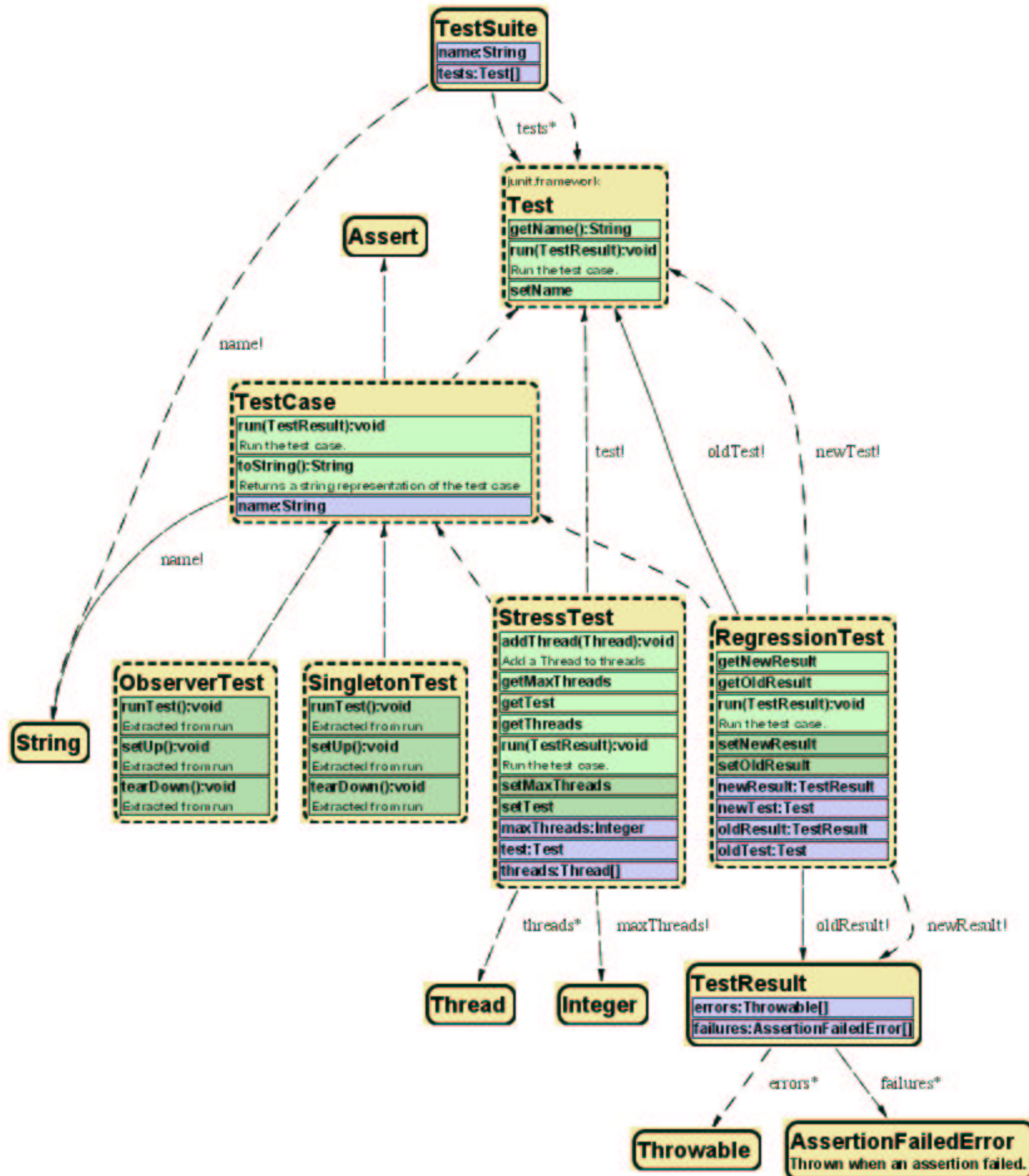

Figure 7-8: A JUnit design alternative with test variants.

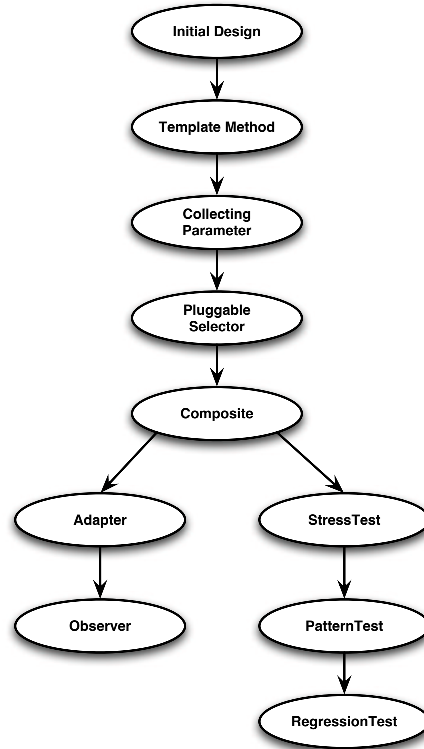

Figure 7-9: The design space of the JUnit framework.

7.2.3 Lessons Learned

Again, several lessons were learned in conducting these redesign experiments with DR. JONES.

- **DR. JONESis particularly good at refactoring frameworks.**

  Frameworks can have very little behavior (as can be seen from the source code in Appendix E), and without it, the only constraints on its design evolution are name conflicts and the visibility of its program elements. These constraints can be easily accounted for within DR. JONES, unlike constraints arising from behavior, which cannot be changed with DR. JONES.

- **DR. JONES doesn't know quite enough about the designs it creates.**

  On the other hand, if we wanted to explore the design of the JUnit framework, and programmers had already used it to implement test cases, these cases would have to be refactored as well. I attempted to evolve the framework along with a simple set of test cases, taken from the currency conversion example described by JUnit's authors (Beck and Gamma, 2000). However, I was unsuccessful, for the following reason.

  Suppose that the initial TestCase class were extended with a MoneyTest class, which has unit test methods for the Money, MoneyBag, and IMoney classes (shown in Figure

7-10). MoneyTest overrides run() in TestCase to run these tests.

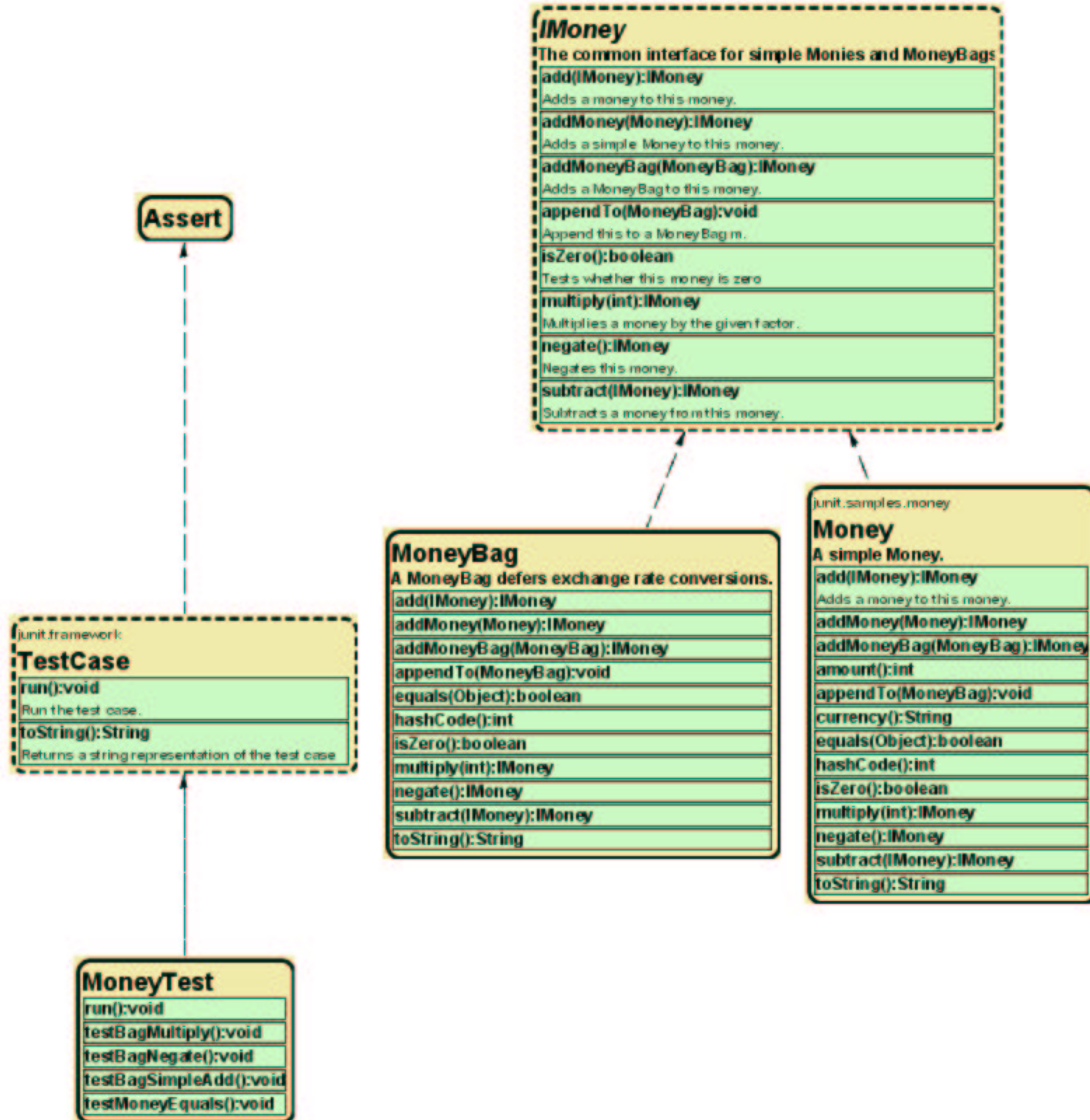

Figure 7-10: MoneyTest implemented in the JUnit framework.

Now we want to refactor the framework class TestCase by decomposing TestCase.run() into TestCase.runTest() (and eventually setUp() and tearDown() as well). DR. JONES does this, then suggests we do the same for the overridden run() in MoneyTest, so that the decomposition is parallel in the class hierarchy.

Unfortunately, as Figure 7-11 shows, DR. JONES then prevents MoneyTest.run() from being decomposed, because it would case a name conflict in TestCase. We know that we want MoneyTest.runTest() to override TestCase.runTest(), but DR. JONES lacks the additional information to link these two methods.

- **The programmer should be able to use design patterns directly.**

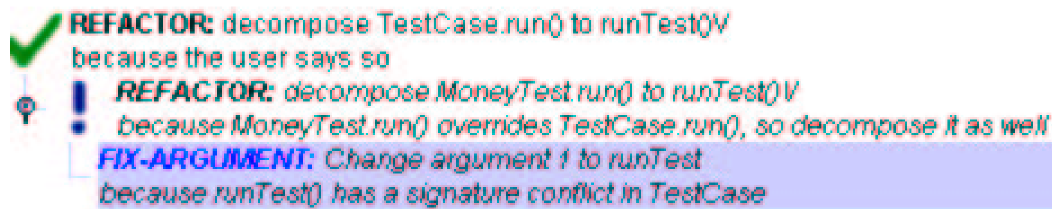

Figure 7-11: Dr. Jones prevents us from decomposing run() in MoneyTest.

Evolving the JUnit framework in the same way as the authors did required manually decomposing the application of design patterns into individual refactoring steps. While not difficult to do by hand, this process could be supported by DR. JONES, which would simplify interaction. Also, DR. JONES would gain another level of abstraction in its design space and enrich its design representation by noting where patterns had been applied. With this additional knowledge, it could assist the programmer in design exploration at the pattern level, and simplify the programmer's view of the design space (as in Figure 7-9).

7.3 Summary

By using DR. JONES on two redesign scenarios, lessons were learned about DR. JONES and its approach to design exploration. Many of these lessons pointed out opportunities to enrich DR. JONES' design representation. The greatest benefit could be gained by giving the representation richer notions of behavior. Adding information about behavior would make it possible to keep the representation accurate across refactorings that redistribute behavior (like Push Down Method and Decompose Method). It could also allow it to more accurately classify a method as a getter, setter, delegate, or other type, so it could be treated appropriately when refactoring. And, it could check some refactoring guards that could not be checked before.

Another way to enrich the representation would be to add knowledge of why certain parts of the design were modified, so that other refactorings could use that context to act appropriately. This could help avoid the problems encountered when refactoring the JUnit framework with already-implemented test cases. Adding levels of abstraction to the design representation would help, too; for example, including design patterns that capture protocols and collaborations among many program elements would enrich the visualization and design exploration capabilities of DR. JONES.

Each of these lessons is an opportunity for future work, which is explored in the next chapter.

Chapter 8

Conclusion

8.1 Thesis

> Program design exploration should – and can – be supported by the computer. Most of software design is redesign, and although programmers frequently redesign software, there is a dearth of tools that help them do it. The tools that do exist primarily automate the transformation of source code, instead of helping the programmer make higher-level design decisions. What is needed is a tool that helps the programmer see and explore the design space of the program.

This research has developed a prototype system, DR. JONES, intended for this purpose. DR. JONES acts like a smart pen-and-paper for the programmer; he has the freedom to leave the source code behind and focus on the important abstractions in the program, while gaining the computer's power to assist the redesign process. DR. JONES uses that power to analyze the program, visualize its design, and provide design assistance. In the background, it keeps track of the design space explored by the programmer.

To achieve the goal of a smart pen-and-paper for redesign, DR. JONES was developed with the following key principles in mind:

- *Support design exploration.* DR. JONES records all designs considered by the programmer, and lets him revisit any prior design to branch off and try a new design alternative.
- *Let the programmer work at the design level.* DR. JONES separates the tasks of design exploration from source transformation. It lets the programmer work with a design-level representation of the program which captures its important abstractions. This representation gives the programmer high-level views of the program, and frees the programmer from worrying about source-level details when redesigning.
- *Give the programmer a variety of design moves.* DR. JONES provides the programmer

with twenty-two refactorings for redesign. I have identified a total of fifty-two refactorings which be incorporated into DR. JONES.

- *Look over the programmer's shoulder and provide design assistance.* As the programmer redesigns, DR. JONES makes suggestions to help him achieve his refactoring goals and make further design improvements to the program.

8.2 Summary of Contributions

To achieve these principles in a working prototype, this research makes the following contributions:

- *It separates the concern of design exploration from source code transformation.* The goal of design exploration is to find an improved design for the program, while the goal of source code transformation is to realize a new design while not introducing new errors into the program. In reality, programmers often cycle between these two levels of maintenance, but current tools do not make a clear distinction between them. This thesis argues the programmer benefits when that distinction is made.

- *It provides an abstract program representation suitable for redesign.* This thesis contributes a design-level representation for Java programs. This design-level representation captures enough information about the program for DR. JONES to give the programmer useful guidance during redesign. It also records the design space the programmer has explored – which design alternatives were considered, and how they are related.

- *It developed a knowledge base of refactorings written specifically for interactive redesign.* Prior work on refactoring has contributed descriptions of them at varying levels of formality and detail (Fowler, 1999; Opdyke, 1992), with the emphasis on the preconditions that ensure their validity and the source code modifications that implement them. This thesis contributes a knowledge base of twenty-two refactorings written specifically for the purpose of interactive redesign.

- *It illustrates a user interface for software design exploration.* DR. JONES provides a novel user interface for the interactive redesign of Java programs. This thesis contributes interface components to render design diagrams at multiple levels of detail, manage the redesign dialogue between the programmer and DR. JONES, and assist the programmer in navigating the design space he has explored.

8.3 Related Work

Several other research projects aim to provide software evolution environments, and DR. JONES touches on research in software visualization, refactoring theory, and Extreme Programming. These related threads of research are discussed here.

8.3.1 Software Evolution Environments

Several past and continuing research projects share DR. JONES' goal of developing an assistant that helps the programmer evolve software. Among them was the Programmer's Apprentice project at the MIT Artificial Intelligence Lab (Rich and Waters, 1987, 1989). The goal of the Programmer's Apprentice was to develop an expert system that would assist the programmer in gathering requirements (Reubenstein, 1990) and designing programs (Shrobe, 1979), and would recognize and compose commonly reused program fragments (clichés) (Wills, 1992), among other capabilities. The Programmer's Apprentice used various general-purpose representations like plans, frames, and logic to reason about programs and support these activities.

DR. JONES shares the Apprentice's vision of an integrated environment that supports many software development activities. However, DR. JONES was deliberately conceived with a more limited task in mind (structural redesign by refactoring), and thus it has more tractable approaches to reverse engineering, representation, and reasoning.

The Star Diagram Browser, Aspect Browser, and related projects are another group of visual redesign systems (Bowdidge and Griswold, 1998; Griswold *et al.*, 1998, 2001). The Star Diagram Browser lets the programmer encapsulate an existing data structures in C and similar languages. The Aspect Browser lets the programmer isolate specific features (aspects) and observe their distribution over the source code. These primarily target C and similar languages, which offer fewer abstraction primitives and thus fewer opportunities for structured redesign (i.e., refactoring). Also, the Star Diagram directly manipulates source code, which DR. JONES does not. However, these tools do address similar visualization issues to DR. JONES.

A promising area of future work is the use of behavioral information to drive structural refactoring. The Daikon invariant extractor uses statistical tests to look for common invariants among subsets of variables in Java programs (Ernst *et al.*, 2001). These invariants can then be used to suggest refactorings, such as removing extraneous parameters and unused return values (Kataoka *et al.*, 2001). Like DR. JONES, Daikon requires little up-front effort to use, and its invariants would be a natural complement to DR. JONES' design instance representation.

8.3.2 Program Understanding and Visualization

Program understanding and visualization tools, while themselves not doing redesign, are crucial in supporting the redesign process. Several research program understanding systems would be helpful to assist redesign.

One approach identifies and visualizes high level software components from lower level structures. Software reflexion models (Murphy *et al.*, 2001; Walker *et al.*, 2000, 1998) allow the programmer to map C procedures and files to implicit, higher-level modules. The Rigi/SHRiMP environment creates nested, hierarchical views of program structures (Storey *et al.*, 1997b; Müller *et al.*, 1992). These views assist redesign by showing high-level design relationships and supporting rapid navigation of the program structure.

Visualization of behavioral information is also important for redesign. Concern Graphs for Java combine structural information with call graphs (Robillard and Murphy, 2002). Visualization of message passing activity among objects and classes can reveal hierarchical decomposition of behavior and degree of coupling (Jerding *et al.*, 1996; Lange and Nakamura, 1995). Historical views of program evolution can show "hot spots" that undergo frequent redesign, feature addition, and feature removal (Lanza and Ducasse, 2002; Eick *et al.*, 2002).

For structural redesign, the detection of pre-existing design patterns is important. Several research prototypes support the recognition and visualization of these patterns (Systä, 2000; Schauer and Keller, 1998; Lange and Nakamura, 1995). Another interesting approach categorizes a class by showing its internal structure, call graph, and local inheritance hierarchy (Lanza and Ducasse, 2001). This reveals patterns like "mute overrider" (overrides methods in its superclass, but does not invoke them).

A wide suite of program views would be a useful complement to DR. JONES, as they enhance sensemaking, flaw detection, and redesign planning. However, it is interesting to ask whether redesign could happen in these other views, or only DR. JONES' own diagrams, which are tailored for redesign. All these systems also struggle with the problem of managing visual complexity; research to filter, abstract, and beautify views of software continues (Rilling *et al.*, 2002; Eichelberger, 2002; Egyed, 2000).

8.3.3 Refactoring Theory

Early on, Parnas advanced the view that a program is a point in a larger design space, and an appropriately designed program can easily move through this space (Parnas, 1979). However, the development of behavior-preserving design moves in design space began in databases, where researchers identified schema transformations that preserved data (Banerjee *et al.*, 1987). Also, early work described schema evolution operations in knowledge bases (Balzer, 1985).

Meaning-preserving transformations began to be identified and automated for pro-

grams as well. Griswold developed a tool for restructuring Scheme programs, and a model for reasoning about how the tool's transformations preserve meaning (Griswold, 1991). The object-oriented programming community also picked up on these ideas and began developing refactorings for object-oriented programs (Johnson and Opdyke, 1993). Out of this work came formal descriptions of refactorings for C++ (Opdyke, 1992) and the first automated source refactoring tool for Smalltalk (Roberts *et al.*, 1997; Roberts, 1999). Fowler compiled the first book-length catalog of refactorings, targeted for C++ and Java (Fowler, 1999).

Other researchers have tried to completely automate refactoring by having the computer choose as well as execute refactorings (Maruyama and Shima, 1999; Moore, 1995; Casais, 1994). However, these approaches often left programs with abstractions and identifiers that were meaningless in the problem domain, confounding program understanding.

Recent work on refactoring theory has focused on refactoring to patterns, that is, applying design patterns to existing programs by refactoring (Zannier and Maurer, 2002; Tokuda and Batory, 2001; Cinne'ide, 2000; Cinne'ide and Nixon, 1999). This technique allows the programmer to make large moves in the program's design space, while maintaining behavior and program correctness. As discussed below, DR. JONES could be extended with this capability through better scripting.

8.3.4 Refactoring Development Environments

Recently, integrated software development environments have begun to incorporate more and more refactorings. Again, these environments operate directly on the source, and may have undo and redo capability but lack design assistance or design space navigation. Examples of development environments with refactorings include TogetherSoft ControlCenter,[1] Rational Rose,[2], IDEA IntelliJ[3], and Eclipse.[4] These environments implement between one and three dozen refactorings, and typically require multiple dialogue windows to complete a single refactoring (as in Figure 8-1).

8.3.5 Extreme Programming

Extreme Programming is a label given to a family of programming philosophies and practices which, taken together, try to simplify software evolution (Beck, 2000). Foremost among them is the aggressive use of refactoring to improve existing programs and adapt them to new requirements. As a corollary, the effort spent in up-front design is minimized,

[1]TogetherSoft, Inc. `http://www.togethersoft.com/`
[2]IBM, Inc. `http://www.rational.com/`
[3]JetBrains, Inc. `http://www.intellij.com/`
[4]Eclipse Project, `http://www.eclipse.org/`

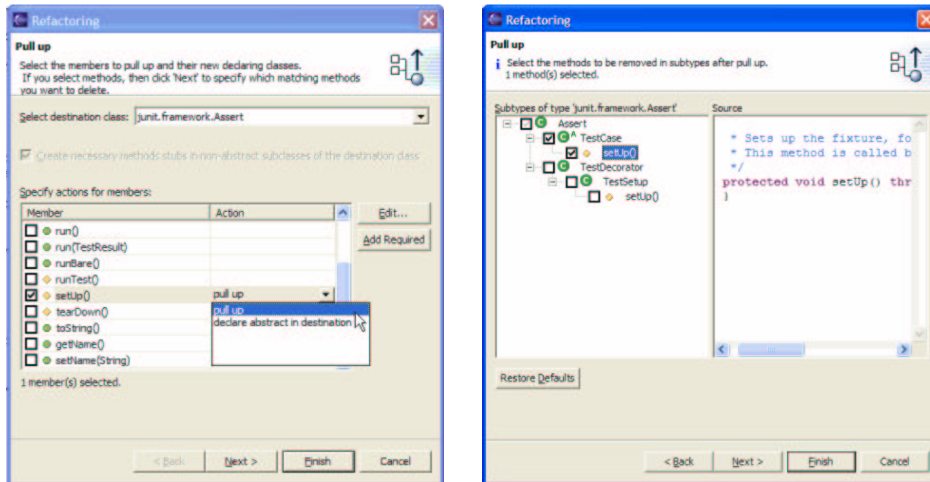

Figure 8-1: Pulling up methods in Eclipse requires detailed, source-level interaction.

as it is better to incrementally evolve a design than to predict all of its required functionality at the beginning. These practices, when useful, become easier to adopt when robust tools support them; DR. JONES is a step in this direction.

8.4 Future Work

DR. JONES is a step in the direction of realizing a truly smart redesign assistant. Ideally, such a tool would act as a very good junior programmer. It wouldn't be able to redesign the program by itself, as that requires having extensive knowledge of the program's problem domain. However, it would be able to interact closely as a redesign assistant. It would help the programmer understand the program, interpret the programmer's redesign intentions (even if vague or inconsistent), and go off to do the dirty work. Most of the repetitive and tedious aspects of software redesign would be automated, freeing the programmer to concentrate on the conceptually difficult parts.

This ideal scenario poses a challenging research agenda, which I outline here. The first part of the agenda addresses some of the lessons learned in using DR. JONES. The second part improves DR. JONES' capabilities as a redesign assistant, within its current framework. The third part is a long-term plan that steps back and lays out steps for creating a truly smart redesign assistant.

8.4.1 Addressing Lessons Learned

The redesign scenarios pointed out some limitations in DR. JONES design representation, which could be addressed by extending it to include both behavioral and context information.

Behavioral information, i.e. what methods actually do, would augment DR. JONES' current structural design representation. The only notion of behavior DR. JONES has now is a collection of Uses... dependencies originating from each method. This could be extended with a lightweight behavioral model – again, simpler than the source code – that would help DR. JONES to more accurately track the evolution of the design when methods are decomposed, and to treat methods appropriately according to the their type (getters, setters, delegates, and so forth). Like DR. JONES' structural representation, any behavioral model added to DR. JONES should be reverse engineered from source code without a lot of effort from the programmer, and more abstract than the source code implementing it.

The second direction is to augment the the representation with role information, that identifies which program elements play roles in larger patterns. These patterns could be the side-effects of individual refactorings, or separate design patterns applied by the programmer in their own right.

One advantage of role information would be to more loosely couple refactorings and suggestions. Instead of a suggestion only being made immediately after a specific refactoring, the refactoring could leave role information in the design representation that would later be used by independent critics. These critics would look for relevant structural patterns, such as partially implemented design patterns, and suggest the programmer complete the pattern with additional refactorings. This situation presents itself in the JUnit scenario: because the Decompose refactoring didn't know it was emulating the Template Method pattern in its superclass, it prevented the programmer from applying it in the subclass.

8.4.2 Broadening Dr. Jones' Refactoring Abilities

As already mentioned, twenty-nine refactorings have been identified as candidates for DR. JONES, but are yet to be implemented in the prototype. With many parts of the refactoring infrastructure available for re-use, there would be no special difficulty in adding them to DR. JONES.

Along with the additional refactorings, support for composing them into reusable scripts would help the programmer make complex design moves made up of many refactorings. For example, a script could be used to apply the Visitor design pattern to existing classes (Tokuda and Batory, 2001). To use it, the programmer would identify the classes to be visited, and the class acting as the visitor. The script would produce the sequence of refactorings that applies the pattern to the classes. Implementing these kinds of scripts in DR. JONES would require extending its scripting facility to include variables and control flow.

Intelligent Visualization. Currently, the programmer can control the content and complexity of DR. JONES' design diagrams by directly manipulating their contents. A better approach would have DR. JONES automatically infer the appropriate contents for a dia-

gram. One hypothesis is that there is a "working set" of program elements involved in recent or likely future refactorings. DR. JONES could infer and visualize this working set, with elements' levels of detail chosen according to their relevance to upcoming refactorings. Because refactorings are local changes (at the design level), this working set should only be a small part of the overall program, and thus diagrams of it shouldn't suffer from excessive visual complexity.

Integrating Design Flaw Diagnosis. DR. JONES helps the programmer plan to fix design flaws, but does not point out the flaws themselves. Some kinds of flaws, dubbed "antipatterns" or "bad smells" (Beck and Fowler, 1999), recur in many programs and contexts, and tools have been developed that detect them (Florjin, 2002). Other flaws can be discovered through code metrics and cyclic dependency checks. DR. JONES could use the output of these analyses to annotate its design diagrams, and provide a starting To Do list of flaws to fix.

Going further, DR. JONES could have a library of redesign plans that match design flaws with refactorings or scripts that ameliorate them. When DR. JONES detects a flaw in its library, it would suggest the corresponding repair plan to the programmer.

Integrating Source Transformation. After the programmer completes a redesign session with DR. JONES, he has has found suitable redesign alternatives, but the work of executing the source code refactorings remains. Because DR. JONES maintains references to the source code locations of program elements and dependencies, it could be extended to do some of this work itself. However, maintaining correct source code locations across refactorings, many of which rearrange parts of the program, poses some difficulty.

An alternative is to link DR. JONES to existing source refactories. Examples of these include RefactorIT, Eclipse, and IntelliJ IDEA. Although none of these implement the full suite of refactorings in the refactoring space yet, they could be extended to do so.

Ideally, the programmer be able explore designs, while the source code is kept updated in the background. The programmer could then move back and forth between the design level and the source code, redesigning at both levels of granularity as he desires.

Adding New Refactorings

Programmers should have the opportunity to add new design moves to DR. JONES, and share useful ones with other programmers. I propose a refactoring specification language, based on the specifications found in Figure 4-3 and Appendix B. The goal of the language is to change DR. JONES from a tool with embedded refactorings to an interpreter whose knowledge can be easily extended. Before developing the interpreter, describing the entire refactoring space in written specifications would ensure that the language has all the required primitives.

8.5 Software Redesign, Broadly Considered

The third part the research agenda a long-term plan which considers a complete environment for redesigning software and capturing the rationale behind the redesign.

8.5.1 Natural Interaction for Redesign

One goal of a redesign tool is to make the redesign process as natural as possible. Our metaphor for "natural" is the programmer using a whiteboard to explain his redesign ideas to another programmer, who can ask questions, make suggestions, and offer feedback. The direct benefit of natural interaction is that it focuses the programmer on the redesign problem, instead of using the redesign tool.

Natural interaction has an important indirect benefit as well – it gives the programmer the opportunity to explain the rationale behind his redesign. After all, programmers don't mind explaining their ideas to other programmers, so capturing (re)design rationale would then become an effortless by-product of redesign. This rationale is crucial for helping the next programmer understand why the program's design evolved in the way it did.

DR. JONES is a step in this direction, but creating a truly natural interface is a challenging problem. Existing research in multimodal interfaces has demonstrated the feasibility of sketch recognition for UML diagrams (Hammond and Davis, 2002), as well as multimodal interaction in other domains (Oviatt *et al.*, 2000; Oltmans, 2000; Cohen *et al.*, 1997). These results are encouraging, but software redesign poses its own unique challenges for multimodal interaction. Some consideration should also be given to the collaborative nature of software design, as complex software is worked on in pairs and teams.

8.5.2 Redesigning Behaviors and Interfaces

Design is more than structure – it encompasses all of the decisions that went into the making of the program. The complement of a program's structure is its behavior, and any significant redesign activity will involve both. Therefore, it would be advantageous for a redesign tool to support redesigning behavior as well as structure.

Unfortunately, behavior is much harder to reason about that structure. Even simple questions – like determining whether two expressions always compute the same value, or the outcome of a conditional – are Turing-complete in the general case. Therefore, less direct approaches must be taken.

One approach maintains a library of common implementation clichés whose behavior is known to be equivalent. The tool can then help the programmer substitute one cliché for another, for example replacing an array-backed list with a linked list. This general approach is taken by the Programmer's Apprentice project (Rich and Waters, 1987). Unfortunately, this approach is limited to the scope of the cliché library, and it is difficult to anticipate all of the clichés needed in advance.

Another approach is to perform lightweight modeling of behavior. Instead of representing the complete behavior of the program, an abstract model captures the program's important constraints and invariants. Ideally, the model and the implementation would be closely synchronized by the redesign tool. When the programmer redesigns the model, the tool would tell him what parts of the source need to be changed. When the programmer redesigns the implementation, the tool would check that it still meets the model's constraints.

Work towards this capability includes the Alloy program modeling language (Jackson, 2000) and the Daikon statistical invariant extractor (Kataoka *et al.*, 2001). Automated software testing is another approach, but modeling has the advantage of being able to describe behavior which hasn't been implemented yet, thus supporting design exploration in the spirit of DR. JONES.

Another aspect of software design is how the program interacts with its user. Again, there are tools for prototyping new user interfaces (Landay and Myers, 1995), but little tool support for evolving existing interfaces to improve their usability or add new features. Like programmers, user interface designers need tools to explore multiple design alternatives and obtain feedback on their ideas.

8.5.3 A New Programming Language

Finally, building a refactoring tool – whether at the design or the source code level – shouldn't be that hard. I suspect that many of the difficulties I encountered in building DR. JONES could be avoided if I were working with a different programming language. It doesn't exist yet, but if I were to invent it, it would support design evolution from the start. It's motto would be, "All Programs Should Be Inherently Easy to Redesign." It would only have features that could be redesigned easily. Because most of software design is redesign, and most of software engineering is maintenance, I believe ease of program evolution is one of the most important criteria for programming language design.

This new language would have its refactoring suite specified and implemented from the start, instead of being done much later than its other tools (compiler, documentation generator, and so forth). These refactorings would be available through a published API, either as part of a library that manipulates static source code, or as part of the language's reflection model. This API would also answer program understanding questions, such as which parts of a program are currently unused. And whenever the interface to a module is refactored, its clients would automatically be refactored as well.

A language with integrated refactorings would free researchers to build higher-level tools that help the programmer redesign and capture the rationale behind the redesign. With these tools, programmers would be able to focus on evolving programs to better match the problems at hand, instead of the repetitive and tedious aspects of program maintenance.

Appendix A

The Refactoring Space

The following table compares three sets of refactorings. The first set are the meaningful refactorings for Java found in the refactoring space (Table 3.2). The second set are the refactorings cataloged in Martin Fowler's book *Refactoring* (Fowler, 1999). The third set are the Smalltalk refactorings implemented in the Smalltalk Refactoring Browser (Roberts *et al.*, 1997).

Because each of these sources uses slightly different names for the same refactorings, the table cross-references equivalent refactorings that appear in more than one source.

The twenty-two refactorings marked with a "†" have been implemented in the DR. JONES prototype.

Several of Fowler's refactorings can be considered a special case or variant of a refactoring in the refactoring space. These refactorings are marked with a "‡."

Table A.1: The Refactoring Space: A Comparison

Refactoring Space	**Fowler's *Refactoring***	**Refactoring Browser**
Create Package		
Create Class†		Create New Class
Create Method†		Add Method
Create Field†		Add Variable
Create Parameter†	Add Parameter	Add Parameter
Remove Package		
Remove Class		Remove Class
Remove Method†	Remove Setting Method‡	
Remove Field		Remove Variable
Remove Parameter	Remove Parameter	
Rename Package†		
Rename Class†		Rename Class
Rename Method†	Rename Method	Rename Method
Rename Field†		Rename Variable
Rename Parameter		
Move Package		
Move Class		
Move Method†	Move Method	Move Method
Move Field	Move Field	Move Variable
Move Parameter		
Hide Class		
Hide Method†	Hide Method	
Hide Field		
Reveal Class†		
Reveal Method†		
Reveal Field		
Compose Packages		
Compose Classes	Inline Class	
Compose Methods		
Decompose Package		
Decompose Class†	Extract Class	
	Extract Interface	
Decompose Method†	Extract Method	
	Separate Query from Modifier‡	
Encapsulate Class		
Encapsulate Method	Replace Method with Method Object	
Encapsulate Field with Accessors†	Encapsulate Field	Create Accessors
	Self Encapsulate Field‡	Abstract Variable

Continued on next page

Continued from previous page

Refactoring Space	**Fowler's *Refactoring***	**Refactoring Browser**
Encapsulate Field to Class†	Encapsulate Collection‡	
	Replace Array (field) with Object‡	
	Replace Data Value (field) with Object‡	
	Replace Record with Data Class‡	
	Replace Type Code with Class‡	
Encapsulate Field to Subclasses†	Replace Type Code with State/Strategy	
Encapsulate Parameters	Introduce Parameter Object	
	Preserve Whole Object‡	
Expose Class	Remove Middle Man	
Expose Method		
Expose Field		
Generalize Class†	Extract Superclass	Create Class
Pull Up Method†	Pull Up Method	Pull Up Method
Pull Up Field	Pull Up Field	Pull Up Variable
Generalize Parameter		
Specialize Class†	Extract Subclass	Create Class
Specialize Class by Type Code†	Replace Type Code with Subclasses	
Push Down Method†	Push Down Method	Push Down Method
Push Down Field	Push Down Field	Push Down Variable
Specialize Parameter		
Alter Type of Field		
Alter Type of Parameter		
52 total	30	16

Appendix B

The Refactoring Knowledge Base

This Appendix includes the content of the refactoring knowledge base of DR. JONES. Each refactoring occupies one entry in the KB, which contains all the information needed to use that refactoring with DR. JONES.

Several predicates and functions are used in the specifications. The predicates are:

- CanSee($class_1$,$class_2$) returns true if $class_2$ is visible to $class_1$. Similarly for CanSee($class$,$method$), CanSee($class$,$field$), CanSee($method$,$method$), and CanSee($method$,$field$).
- HasField($class$,$field$) returns true if $field$ is a member of $class$. Similarly for HasMethod($class$,$method$) and HasParameter($method$,$parameter$).
- IsAbstract($method$) returns true if $method$ is declared abstract. Similarly for IsAbstract($class$).
- IsClass($type$) returns true if $type$ is a reference type.
- IsConstructor($method$) returns true if $method$ is a constructor.
- IsFinal($field$) returns true if $field$ is declared final.
- IsInterface($class$) returns true if $class$ is an interface.
- IsStatic($method$) returns true if $method$ is declared static.
- IsSubClass($class_1$,$class_2$) returns true if $class_1$ is a direct or indirect subclass of $class_2$. Similarly for IsSuperClass($class_1$,$class_2$).
- LegalIdentifier($name$) returns true if $name$ is a legal Java identifier.
- NoNameConflict($className$, $package$) returns true if no other class in $package$ is named $className$. Similarly for NoNameConflict($methodName$, $methods$) and NoNameConflict($fieldName$, $fields$).

- NoSignatureConflict($method$,$class$) returns true if no other method in $class$ has the same signature as $method$.
- Overloads($method_1$,$method_2$) returns true if $method_1$ overloads $method_2$, and $method_1$ and $method_2$ are members of the same class.
- OverridesAny($method_1$,$method_2$) returns true if $method_2$ overrides $method_1$ directly or indirectly, or vice versa.
- Reads($method$, $field$) returns true if $method$ reads the value of $field$. Similarly for Writes($method$, $field$).
- SimilarName($name_1$,$name_2$) returns true if $name_1$ is a substring of $name_2$.
- StandardClassName($className$) returns true if $className$ follows conventions for naming classes in Java. Similarly for StandardPackageName, StandardMethodName, StandardFieldName, and StandardParameterName.

The following functions return values or sets of values.

- ClassOf($type$) returns the class of the reference type $type$.
- MakeGetter($field$) returns a new field that returns the value of $field$. Similarly for MakeSetter($field$).
- MethodsOf($class$) returns the set of methods in $class$. Similarly for FieldsOf($class$).
- NameOf($element$) returns the name of $element$.
- PackageOf($class$) returns the package of $class$.

The following functions directly modify the design representation.

- CopySignature($method_1$, $method_2$) copies the Accepts, Returns, and Throws dependencies from $method_1$ to $method_2$, and sets $method_2$ to have the same modifiers as $method_1$.
- CopyUses($method_1$, $method_2$) copies the UsesClass, UsesMethod, and UsesField dependencies from $method_1$ to $method_2$.

Create *in* aPackage *a class named* aClassName

Must Guards	
M1	LegalIdentifier(aClassName) $\hookleftarrow$ Modify argument aClassName
M2	NoNameConflict(aClassName, aPackage) $\hookleftarrow$ Modify argument aClassName

Should Guards	
S1	StandardClassName(aClassName) $\hookleftarrow$ Modify argument aClassName
S2	$\forall package$ NoNameConflict(aClassName, $package$) $\hookleftarrow$ Modify argument aClassName

Design Transforms	
X1	+Class(aClassName, aPackage)

Create *in* aClass *a method named* aMethodName *returning* aType

Let aMethod be the method created.

	Must Guards
M1	LegalIdentifier(aMethodName) $\hookleftarrow$ Modify argument aMethodName
M2	$\forall c$ IsSuperClass(c, aClass) $\rightarrow$ NoSignatureConflict($aMethod$, c) $\hookleftarrow$ Modify argument aMethodName
M3	IsClass(aType) $\rightarrow$ CanSee(aClass, ClassOf(aType)) $\hookleftarrow$ Reveal ClassOf(aType)

	Should Guards
S1	StandardMethodName(aMethodName) $\hookleftarrow$ Modify argument aMethodName
S2	$\forall c$ IsSubClass(c, aClass) $\rightarrow$ NoSignatureConflict($aMethod$, c) $\hookleftarrow$ Remind user to check that aMethod will not be improperly hidden in c

	Design Transforms
X1	$M \leftarrow$ +Method(aMethodName, aClass)
X2	$M.returnType \leftarrow$ aType
X3	$M.abstract \leftarrow$ IsAbstract(aClass)
X4	$M.static \leftarrow \bot$
X5	$M.access \leftarrow$ aClass.$access$

Create *in* aClass *a field named* aFieldName *with type* aType *and multiplicity* aMult

Must Guards	
M1	LegalIdentifier(aFieldName) $\hookleftarrow$ Modify argument aFieldName
M2	IsClass(aType) $\rightarrow$ CanSee(aClass, ClassOf(aType)) $\hookleftarrow$ Reveal ClassOf(aType)
M3	$\forall c$ IsSuperClass(c, aClass) $\rightarrow$ NoNameConfict(aFieldName, FieldsOf(c)) $\hookleftarrow$ Modify argument aFieldName

Should Guards	
S1	StandardFieldName(aFieldName) $\hookleftarrow$ Modify argument aFieldName
S2	$\forall c$ IsSubClass(c, aClass) $\rightarrow$ NoNameConflict(aFieldName, FieldsOf(c)) $\hookleftarrow$ Remind user to check that aFieldName will not be improperly hidden in c

Design Transforms	
X1	$F \leftarrow$ +Field(aFieldName, aClass)
X2	$F.type \leftarrow$ aType, if aMult $= \neg$ *or* aMult $= +$
X3	$F.type \leftarrow$ aType[], if aMult $= *$
X4	$F.access \leftarrow Private$
X5	$F.static \leftarrow \bot$
X6	+HasA(F, ClassOf(aType), aMult), if IsClass(aType)

Design Suggestions	
R1	$\top$ $\rightarrow$ Encapsulate F *with accessor methods*

Create *for* aMethod *a parameter named* aParamName *with type* aType

Let aMethod′ be aMethod with the new parameter added.

Must Guards	
M1	LegalIdentifier(aParamName) $\hookleftarrow$ Modify argument aParamName
M2	$\forall p$ HasParameter(aMethod, p) $\rightarrow$ NameOf(p) $\neq$ aParamName $\hookleftarrow$ Modify argument aParamName
M3	IsClass(aType) $\rightarrow$ CanSee(ClassOf(aMethod), ClassOf(aType)) $\hookleftarrow$ Reveal ClassOf(aType)
M4	$\forall c$ IsSuperClass(c, ClassOf(aMethod)) $\rightarrow$ NoSignatureConflict(aMethod′, c) $\hookleftarrow$ Modify argument aParamName

Should Guards	
S1	StandardParameterName(aParamName) $\hookleftarrow$ Modify argument aParamName
S2	$\forall c$ IsSubClass(c, ClassOf(aMethod)) $\rightarrow$ NoSignatureConflict(aMethod′, c) $\hookleftarrow$ Remind user to check that aMethod′ will not be improperly hidden in c

Design Transforms	
X1	$P \leftarrow$ +Parameter(aParamName, aMethod)
X2	$P.type \leftarrow$ aType

Remove aMethod

Must Guards	
M1	$\neg\, \exists method$ UsesMethod($method$, aMethod)
M2	$\neg\, \exists method$ Overrides($method$, aMethod)

Design Transforms	
X1	−aMethod

Rename aPackage to aPackageName

Must Guards	
M1	aPackageName $\neq$ NameOf(aPackage)
M2	LegalIdentifier(aPackageName) $\hookleftarrow$ Modify argument aPackageName
M3	$\forall package$ aPackageName $\neq$ NameOf($package$) $\hookleftarrow$ Modify argument aPackageName

Should Guards	
S1	StandardPackageName(aPackageName) $\hookleftarrow$ Modify argument aPackageName

Design Transforms	
X1	aPackage.$name$ $\leftarrow$ aPackageName

Rename aClass to aClassName

Must Guards	
M1	aClassName $\neq \text{NameOf}($aClass$)$
M2	$\text{LegalIdentifier}($aClassName$)$ $\hookleftarrow$ Modify argument aClassName
M3	$\text{NoNameConflict}($aClassName$,\ aPackage)$ $\hookleftarrow$ Modify argument aClassName

Should Guards	
S1	$\text{StandardClassName}($aClassName$)$ $\hookleftarrow$ Modify argument aClassName
S2	$\forall package\ \text{NoNameConflict}($aClassName$,\ package)$ $\hookleftarrow$ Modify argument aClassName

Design Transforms	
X1	aClass$.name \ \leftarrow$ aClassName

Design Suggestions	
R1	$\exists class\ \text{SimilarName}(\text{NameOf}(class),\ \text{NameOf}($aClass$))\ \wedge\ (class \neq$ aClass$)$ $\rightarrow$ Rename $class$

Rename aMethod to aMethodName

Must Guards	
M1	aMethodName $\neq$ NameOf(aMethod)
M2	$\neg$IsConstructor(aMethod)
M3	LegalIdentifier(aMethodName) $\hookleftarrow$ Modify argument aMethodName
M4	$\forall c$ IsSuperClass(c, ClassOf(aMethod)) $\rightarrow$ NoSignatureConflict(aMethodName, c) $\hookleftarrow$ Modify argument aMethodName
M5	$\forall c$ IsSubClass(c, ClassOf(aMethod)) $\rightarrow$ NoSignatureConflict(aMethodName, c) $\hookleftarrow$ Modify argument aMethodName

Should Guards	
S1	StandardMethodName(aMethodName) $\hookleftarrow$ Modify argument aMethodName

Design Transforms	
X1	aMethod.$name$ $\leftarrow$ aMethodName
X2	$\forall method$ $method.name$ $\leftarrow$ aMethodName, if OverridesAny($method$, aMethod)

Design Suggestions	
R1	$\forall method$ Overloads($method$, aMethod) $\rightarrow$ Rename $method$ aMethodName

Rename aField *to* aFieldName

Must Guards	
M1	aFieldName $\neq$ NameOf(aField)
M2	LegalIdentifier(aFieldName) $\hookleftarrow$ Modify argument aFieldName
M3	$\forall c$ IsSuperClass(c, ClassOf(aField)) $\rightarrow$ NoNameConfict(aFieldName, FieldsOf(c)) $\hookleftarrow$ Modify argument aFieldName
M4	$\forall c$ IsSubClass(c, ClassOf(aField)) $\rightarrow$ NoNameConflict(aFieldName, FieldsOf(c)) $\hookleftarrow$ Modify argument aFieldName

Should Guards	
S1	StandardFieldName(aFieldName) $\hookleftarrow$ Modify argument aFieldName

Design Transforms	
X1	aField.$name$ $\leftarrow$ aFieldName

Move aMethod *from* oldClass *to* newClass *via* aField

Must Guards	
M1	oldClass $\neq$ newClass
M2	$\neg$IsStatic(aMethod)
M3	$\neg$IsConstructor(aMethod)
M4	$\neg$IsInterface(oldClass)
M5	IsInterface(newClass) $\rightarrow$ IsAbstract(aMethod)
M6	$\neg$IsSuperClass(oldClass, newClass)
M7	$\neg$IsSubClass(oldClass, newClass)
M8	$aVia.type$ = newClass $\wedge$ $aVia.mult$ = !
M9	CanSee(oldClass, newClass) $\hookleftarrow$ Reveal newClass
M10	aMethod.$access$ $\neq$ private $\hookleftarrow$ Reveal aMethod
M11	$\forall method$ UsesMethod(aMethod, $method$) $\rightarrow$ CanSee(newClass, $method$) $\hookleftarrow$ Reveal *method*
M12	$\forall method$ UsesField(aMethod, $field$) $\rightarrow$ CanSee(newClass, $field$) $\hookleftarrow$ Reveal *field*
M13	$\forall field$ UsesField(aMethod, $field$) $\rightarrow$ $\neg$HasField(oldClass, $field$) $\hookleftarrow$ Encapsulate *field*
M14	$\forall field$ UsesField(aMethod, $field$) $\rightarrow$ CanSee(newClass, $field$) $\hookleftarrow$ Reveal *field*

Should Guards	
S1	$\forall class$ SubClassOf(c, newClass) $\rightarrow$ NoNameConflict(NameOf(aMethod), $class$) $\hookleftarrow$ Remind user to check that aMethod is not overloaded by a method in c

Design Transforms	
X1	$extract$ aMethod $from$ oldClass
X2	$insert$ aMethod $into$ newClass
X3	$Delegate$ $\leftarrow$ +Method(NameOf(aMethod), oldClass)
X4	+UsesMethod($Delegate$, aMethod)
X5	+UsesField($Delegate$, aField)

Design Suggestions	
R1	$\forall method$ Overrides($method$, aMethod) $\rightarrow$ Move *method to a subclass of* newClass
R2	$\forall method$ Overrides(aMethod, $method$) $\rightarrow$ Move *method to a superclass of* newClass
R3	$\forall method$ Overloads($method$, aMethod) $\rightarrow$ Move *method to* newClass

Hide aMethod *with* narrowerModifier

Let aMethod′ be aMethod with the narrowModifier.

Must Guards	
M1	aMethod.$access \neq$ private
M2	$\neg$IsInterface(ClassOf(aMethod))
M3	$\forall method$ UsesMethod($method$, aMethod) $\rightarrow$ CanSee($method$, aMethod′)
M4	$\neg\ \exists method$ OverridesAny($method$, aMethod)

Design Transforms	
X1	aMethod.$access \leftarrow narrowerModifier$

Reveal aClass *with* widerModifier

Must Guards
M1 $\neg$IsPublic(aClass)

Design Transforms
X1 aClass.*access* $\leftarrow$ widerModifier

Reveal aMethod *with* widerModifier

Must Guards	
M1	aMethod.$access \neq$ public

Design Transforms	
X1	aMethod.$access$ $\leftarrow$ widerModifier

Decompose aClass *into a class named* aClassName

Must Guards	
M1	LegalIdentifier(aClassName) $\hookleftarrow$ Modify argument aClassName
M2	NoNameConflict(aClassName, $aPackage$) $\hookleftarrow$ Modify argument aClassName
M3	$\forall c$ IsSuperClass(c, aClass) $\rightarrow$ NoNameConflict(aClassName, FieldsOf(c)) $\hookleftarrow$ Modify argument aClassName

Should Guards	
S1	StandardClassName(aClassName) $\hookleftarrow$ Modify argument aClassName
S2	$\forall package$ NoNameConflict(aClassName, $package$) $\hookleftarrow$ Modify argument aClassName
S3	$\forall c$ IsSubClass(c, aClass) $\rightarrow$ NoNameConflict(aClassName, FieldsOf(c)) $\hookleftarrow$ Remind user to check that aClassName will not be improperly hidden in c

Design Transforms	
X1	$C \leftarrow$ +Class(aClassName, $package$Of(aClass))
X2	$F \leftarrow$ +Field(aClassName, aClass)
X3	$C.abstract \leftarrow$ aClass.$abstract$
X4	$F.type \leftarrow C$
X5	$F.access \leftarrow Private$
X6	$F.static \leftarrow \bot$
X7	+HasA(F, C, !)

Design Suggestions	
R1	$\top$ $\rightarrow$ Encapsulate F *with accessor methods*

Decompose aMethod *into a method named* aMethodName *returning* aType

	Must Guards
M1	LegalIdentifier(aMethodName) $\hookleftarrow$ Modify argument aMethodName
M2	$\forall c$ IsSuperClass($c, aClass$) $\rightarrow$ NoSignatureConflict(aMethodName, c) $\hookleftarrow$ Modify argument aMethodName
M3	IsClass(aType) $\rightarrow$ CanSee($aClass$, ClassOf(aType)) $\hookleftarrow$ Reveal ClassOf(aType)

	Should Guards
S1	StandardMethodName(aMethodName) $\hookleftarrow$ Modify argument aMethodName
S2	$\forall c$ IsSubClass($c, aClass$) $\rightarrow$ NoSignatureConflict(aMethodName, c) $\hookleftarrow$ Remind user to check that aMethodName will not be improperly hidden in c

	Design Transforms
X1	$M \leftarrow$ +Method(aMethodName, $aClass$)
X2	CopySignature(aMethod, M)
X3	CopyUses(aMethod, M)

	Design Suggestions
R1	$\forall method$ Overrides($method$, aMethod) $\rightarrow$ Decompose *method into a method named* aMethodName

Encapsulate aField *with accessor methods*

Let G and S be the getter and setter methods created.

	Must Guards
M1	$\neg$IsStatic(aField)
M2	$\neg$IsFinal(aField)
M3	$\forall c$ IsSuperClass(c, ClassOf(aField)) $\rightarrow$ NoSignatureConflict(G, c)
M4	$\forall c$ IsSuperClass(c, ClassOf(aField)) $\rightarrow$ NoSignatureConflict(S, c)

	Should Guards
S1	$\forall c$ IsSubClass(c, ClassOf(aField)) $\rightarrow$ NoSignatureConflict(G, c) $\hookleftarrow$ Remind user to check that G will not be improperly hidden in c
S2	$\forall c$ IsSubClass(c, ClassOf(aField)) $\rightarrow$ NoSignatureConflict(S, c) $\hookleftarrow$ Remind user to check that S will not be improperly hidden in c

	Design Transforms
X1	$G \leftarrow$ +MakeGetter(aField);
X2	$S \leftarrow$ +MakeSetter(aField);
X3	+UsesField(G, aField)
X4	+UsesField(S, aField)
X5	+Returns(G, ClassOf(aField.$type$), if IsClass(aField.$type$)
X6	+Accepts(S, ClassOf(aField.$type$), if IsClass(aField.$type$)
X7	$\forall method$ + UsesMethod($method$, G), if Reads($method$, aField)
X8	$\forall method$ − UsesField($method$, aField), if Reads($method$, aField)
X9	$\forall method$ + UsesMethod($method$, S), if Writes($method$, aField)
X10	$\forall method$ − UsesField($method$, aField), if Writes($method$, aField)
X11	aField.$access$ $\leftarrow$ private

Encapsulate aTypeField *to a class with subclasses* $\text{valueField}_1 \ \ldots \ \text{valueField}_n$

	Must Guards
M1	$\text{NoNameConflict}(\text{NameOf}(\text{aTypeField}),\ \text{PackageOf}(\text{ClassOf}(\text{aTypeField})))$ $\hookleftarrow$ Rename aTypeField
M2	$\forall i\ \text{NoNameConflict}(\text{NameOf}(\text{valueField}_i),\ \text{PackageOf}(\text{ClassOf}(\text{aTypeField})))$ $\hookleftarrow$ Rename valueField_i
M3	$\forall i\ \text{aTypeField}.type\ =\ \text{valueField}_i.type$
M4	$\text{IsFinal}(\text{aTypeField})$
M5	$\forall i\ \text{IsFinal}(\text{valueField}_i)$
M6	$\neg\exists method\ \text{UsesField}(method,\ \text{aTypeField})\ \wedge\ (\text{ClassOf}(method) \neq \text{ClassOf}(\text{aTypeField}))$
M7	$\neg\exists subClass\ \text{IsA}(subClass,\ \text{ClassOf}(\text{aTypeField}))$
M8	$\perp$ $\hookleftarrow$ Remind user to check that aTypeField only takes values from $\text{valueField}_1 \ \ldots \ \text{valueField}_n$

	Should Guards
S1	$\forall i\ \forall package\ \text{NoNameConflict}(\text{NameOf}(\text{aTypeField}),\ package)$ $\hookleftarrow$ Rename aTypeField
S2	$\forall i\ \forall package\ \text{NoNameConflict}(\text{NameOf}(\text{valueField}_i),\ package)$ $\hookleftarrow$ Rename valueField_i

	Design Transforms
X1	$C\ \leftarrow\ +\text{Class}(\text{NameOf}(\text{aTypeField}),\ \text{PackageOf}(\text{ClassOf}(\text{aTypeField})))$
X2	$\forall i\ C_i\ \leftarrow\ +\text{Class}(\text{NameOf}(\text{valueField}_i),\ \text{PackageOf}(\text{ClassOf}(\text{aTypeField})))$
X3	$\forall i\ +\text{IsA}(C_i,\ C)$
X4	$\forall i\ -\text{valueField}_i$
X5	$\text{aTypeField}.type\ \leftarrow\ C$
X6	$+\text{HasA}(\text{aTypeField},\ C,\ \neg)$

	Design Suggestions
R1	$\neg\text{StandardClassName}(C)$ $\rightarrow$ Rename C
R2	$\forall i\ \neg\text{StandardClassName}(C_i)$ $\rightarrow$ Rename C_i
R3	$\forall i\ \forall method\ \text{UsesField}(method,\ \text{valueField}_i)$ $\rightarrow$ Move *method to C and* Specialize *method to* $C_1\ \ldots\ C_n$

Generalize $\mathsf{class_1} \ldots \mathsf{class_n}$ *to a superclass named* aClassName

Must Guards	
M1	$\mathrm{LegalIdentifier}(\mathsf{aClassName})$ $\hookleftarrow$ Modify argument aClassName
M2	$\mathrm{NoNameConflict}(\mathsf{aClassName},\ \mathrm{PackageOf}(class_1))$ $\hookleftarrow$ Modify argument aClassName

Should Guards	
S1	$\mathrm{StandardClassName}(\mathsf{aClassName})$ $\hookleftarrow$ Modify argument aClassName
S2	$\forall package\ \mathrm{NoNameConflict}(\mathsf{aClassName},\ package)$ $\hookleftarrow$ Modify argument aClassName
S3	$\forall i\ \mathrm{PackageOf}(\mathsf{class_i}) = \mathrm{PackageOf}(class_1)$ $\hookleftarrow$ Modify argument $\mathsf{Moveclass_i toPackageOf(class_1)}$

Design Transforms	
X1	$C \leftarrow +\mathrm{Class}(\mathsf{aClassName},\ \mathrm{PackageOf}(class_1))$
X2	$C.access \leftarrow Public$
X3	$C.static \leftarrow \bot$
X4	$C.abstract \leftarrow \top,\ if\ \mathrm{IsAbstract}(\mathsf{class_i})\ for\ any\ i$
X5	$C.abstract \leftarrow \bot,\ if\ \neg\mathrm{IsAbstract}(\mathsf{class_i})\ for\ all\ i$
X6	$\forall i\ +\mathrm{IsA}(\mathsf{class_i},\ C)$

Design Suggestions	
R1	$\exists method\ \forall i\ \mathrm{SignatureConflicts}(method,\ \mathsf{class_i})$ $\rightarrow$ Pull Up *method to* C

Pull Up $\mathsf{method}_1 \ \ldots \ \mathsf{method}_n$ *to* aSuperClass

	Must Guards
M1	$\forall i \ \neg\text{NoSignatureConflict}(\mathsf{method_i},\ \mathsf{method_1})$
M2	$\forall i \ \text{IsA}(\text{ClassOf}(\mathsf{method_i}),\ \mathsf{aSuperClass})$
M3	$\forall i \ \neg\text{IsStatic}(\mathsf{method_i})$
M4	$\forall i \ \neg\text{IsConstructor}(\mathsf{method_i})$
M5	$\forall i \ \mathsf{method_i}.access \neq \mathsf{private}$ $\hookleftarrow$ Reveal $\mathsf{method_i}$
M6	$\forall i \ \neg\exists method \ \text{IsMethodOf}(method,\ \mathsf{aSuperClass}) \ \wedge \ \text{SignatureConflicts}(method,\ \mathsf{method_1})$ $\hookleftarrow$ Rename *method*

	Design Transforms
X1	$M \ \leftarrow \ +\text{Method}(\text{NameOf}(\mathsf{method_1}),\ \mathsf{aSuperClass})$
X2	$\text{CopySignature}(\mathsf{method_1},\ M);$
X3	$\text{CopyUses}(\mathsf{method_1},\ M);$
X4	$M.abstract \ \leftarrow \ \top,\ if \ any \ of$ $\mathsf{method_i}$ *are abstract*
X5	$M.abstract \ \leftarrow \ \bot,\ if \ none \ of$ $\mathsf{method_i}$ *are abstract*
X6	$\forall i \ + \text{Overrides}(\mathsf{method_i},\ M)$

	Design Suggestions
R1	$\forall method \ \text{Overloads}(method,\ \mathsf{method_1})$ $\rightarrow$ Pull Up *method to* aSuperClass

Specialize aClass *to a class named* aClassName

Must Guards	
M1	$\neg$IsFinal(aClass)
M2	LegalIdentifier(aClassName) $\hookleftarrow$ Modify argument aClassName
M3	NoNameConflict(aClassName, PackageOf(aClass)) $\hookleftarrow$ Modify argument aClassName

Should Guards	
S1	StandardClassName(aClassName) $\hookleftarrow$ Modify argument aClassName
S2	$\forall$*package* NoNameConflict(aClassName, *package*) $\hookleftarrow$ Modify argument aClassName

Design Transforms	
X1	$C \leftarrow$ +Class(aClassName, PackageOf(aClass))
X2	+IsA(C, aClass)

Specialize aTypeField *to subclasses named* $\mathsf{valueField}_1 \ldots \mathsf{valueField}_n$

Must Guards	
M1	$\forall i$ NoNameConflict(NameOf($\mathsf{valueField}_i$), PackageOf(ClassOf(aTypeField))) $\hookleftarrow$ Rename $\mathsf{valueField}_i$
M2	$\forall i$ aTypeField.$type$ = $\mathsf{valueField}_i$.$type$
M3	IsFinal(aTypeField)
M4	$\forall i$ IsFinal($\mathsf{valueField}_i$)
M5	$\neg\exists method$ Uses($method$, aTypeField) $\wedge$ (ClassOf($method$) $\neq$ ClassOf(aTypeField))
M6	$\neg\exists subClass$ IsA($subClass$, ClassOf(aTypeField))

Should Guards	
S1	$\perp$ $\hookleftarrow$ Remind user to check that aTypeField only takes values from $\mathsf{valueField}_1 \ldots \mathsf{valueField}_n$
S2	$\forall i\ \forall package$ NoNameConflict(NameOf($\mathsf{valueField}_i$), $package$) $\hookleftarrow$ Rename $\mathsf{valueField}_i$

Design Transforms	
X1	$\forall i\ C_i \leftarrow$ +Class(NameOf($\mathsf{valueField}_i$), PackageOf(ClassOf(aTypeField)))
X2	$\forall i$ + IsA(C_i, ClassOf(aTypeField))
X3	$\forall i\ - \mathsf{valueField}_i$
X4	−aTypeField

Design Suggestions	
R1	$\forall method$ Uses($method$, aTypeField) $\rightarrow$ Push Down *method to subclasses* $C_1 \ldots C_n$
R2	$\forall i\ \neg$StandardClassName(C_i) $\rightarrow$ Rename C_i

Push Down aMethod *to subclasses* $\mathsf{class}_1 \ldots \mathsf{class}_n$

Must Guards	
M1	$\neg$IsConstructor(aMethod)
M2	$\neg$IsStatic(aMethod)
M3	aMethod.*access* $\neq$ private
M4	$\forall i$ IsA(class_i, ClassOf(aMethod))
M5	$\forall i$ $\neg$IsInterface(class_i)
M6	$\forall i$ NoSignatureConflict(aMethod, class_i) $\hookleftarrow$ Rename aMethod

Should Guards	
S1	$\forall i$ NoNameConflict(NameOf(aMethod), class_i) $\hookleftarrow$ Rename aMethod

Design Transforms	
X1	$Ai\ M_i \leftarrow$ +Method(NameOf(aMethod), class_i)
X2	Ai CopySignature(aMethod, M_i)
X3	Ai CopyUses(aMethod, M_i)
X4	$\forall i$ + Overrides(M_i, aMethod)

Design Suggestions	
R1	$\forall method$ Overloads($method$, aMethod) $\rightarrow$ Push Down *method to* $\mathsf{class}_1, \ldots, \mathsf{class}_n$

Appendix C

A Refactoring Implementation

This appendix contains the source code implementing the Specialize aTypeCodeField refactoring. The bulk of the work is done by the evalMust(), evalShould(), apply(DInstance, DInstance), evalDesign(), and evalSource() methods, which implement the five parts of the knowledge base entry for the refactoring.

```
package edu.mit.infoarch.drjones.refactor;

import java.util.HashSet;
import java.util.List;

import edu.mit.infoarch.drjones.tg.DJClass;
import edu.mit.infoarch.drjones.tg.DJField;
import edu.mit.infoarch.drjones.tg.DJMethod;
import edu.mit.infoarch.drjones.tg.DJPackage;
import edu.mit.infoarch.drjones.tg.DepSet;
import edu.mit.infoarch.drjones.tg.DependencyType;
import edu.mit.infoarch.drjones.tg.EltSet;
import edu.mit.infoarch.drjones.tg.IsA;
import edu.mit.infoarch.drjones.tg.DInstance;
import edu.mit.infoarch.drjones.tg.UsesField;
import edu.mit.infoarch.util.Monad;
import edu.mit.infoarch.util.Util;
import java.util.Set;



/**
 * A refactoring which breaks a class into subclasses
 * based on the values of a type code field.
 *
 * @author    <a href="mailto:mfoltz@ai.mit.edu">mark a. foltz</a>
 * @version   $Id$
 *
 * @file      <tt>/home/ai3/mfoltz/drjones/refactor/SpecTypeCode.java</tt>
 * @created   Wed Apr  9 17:56:04 2003
 * @copyright Copyright (C) 2003 by mark a. foltz
 */

public class SpecTypeCode extends Refactoring
{

  private  DJField s_code;
  private  EltSet s_values;
  private  DJClass s_class;
  private  DJPackage s_package;

  private  DepSet s_codeUses;

  private DJField t_code;
  private  EltSet t_newClasses;

  private SpecTypeCode(DJField aCode, EltSet values)
  {
    super();
    s_code = aCode;
    s_class = s_code.getContainingClass();
    s_codeUses = s_code.incoming(DependencyType.USES_FIELD);
    s_package = s_class.getContainingPackage();
    s_values = values;
    t_newClasses = new EltSet();
  }

  SpecTypeCode()
  {
```

```
}

public String getName()
{
  return "SpecializeTypeCode";
}


/** Evaluate the must-guards. */
protected void evalMust(final DInstance s)
{

  addMust(!s_values.isEmpty(), "no value fields specified!");

  addMust(s_code.isFinal(),s_code+" is not final");

  s_values.map(new Monad() {
      public void f(Object x)
      {
        DJField f = (DJField) x;
        addMust(f.isFinal(),
                f+" cannot be a type value because is not final");
        addMust(f.getFieldType().equals(s_code.getFieldType()),
                f+" cannot be a type value \
                because its type does not match "+s_code);
        addMust(gNotAmbiguous(f.name(), s_package.classes(),
                              aRef(RefVerb.RENAME, f,
                                   f.name()+
                                   " is ambiguous in "+
                                   s_package)));
      }
    });

  addMust(s_class.subClassSet().size() == 1,
          s_class+" already has subclasses");

  s_codeUses.projectFrom().map(new Monad() {
      public void f(Object x)
      {
        DJMethod m = (DJMethod) x;
        EltSet all = s_class.methods().toMutable();
        all.add(s_class.constructors());
        addMust(all.contains(m),
                s_code+" is used by method "+m+" not in "+s_class);
      }
    });

  addMust(true, "Check that "+s_code+" is only assigned values "+
          s_values, "Dr. Jones cannot check this by itself");

}

/** Evaluate the should-guards. */
protected void evalShould(final DInstance s)
{
  s_values.map(new Monad() {
      public void f(Object x)
      {
        DJField f = (DJField) x;
```

```
        addShould(gNotAmbiguous(f.name(), s.classes(),
                                aRef(RefVerb.RENAME, f,
                  f.name()+" is ambiguous in some package")));
      }
    });

}

/** Apply the refactoring to the new design instance t. */
protected void apply(final DInstance s, final DInstance t)
{
  final DJClass t_class = t.lookupClass(s_class);
  final DJPackage t_package = t.lookupPackage(s_package);

  t_code = t.lookupField(s_code);

  t.remove(t_code);

  s_values.map(new Monad() {
      public void f(Object x)
      {
        DJField value = (DJField) x;
        DJClass valueClass = new DJClass(value.getName(), false);
        t_newClasses.add(valueClass);

        // assert the new is-a dependency
        t.put(new IsA(valueClass,t_class));

        // insert into package, fixing up dependencies as we go along
        t.insertClass(valueClass, t_package);

        // remove the old field
        t.remove(t.lookupField(value));
      }
    });

}

/** Evaluate the design suggestions. */
protected void evalDesign(final DInstance t)
{

  s_codeUses.projectFrom().map(new Monad() {
      public void f(Object x)
      {
        DJMethod m = (DJMethod) x;
        if (!(m.isConstructor() || m.isDelegate())) {
          addRec(RefVerb.SPECIALIZE, m, t_newClasses,
                 m+"'s behavior depends on the type code field");
        }
      }
    });

  t_newClasses.map(new Monad() {
      public void f(Object x)
      {
        DJClass c = (DJClass) x;
        if (!DJClass.isLegalName(c.name())) {
          addRec(RefVerb.RENAME, c, DJClass.standardizeName(c.name()),
```

```
                    c.name()+" is an unusual class name");
              }
            }
          });

      }

      /** Evaluate the source editing instructions. */
      protected void evalSource(DInstance s)
      {
        addSource(s_code,"Remove this field");
        s_values.map(new Monad() {
            public void f(Object x)
            {
              DJField f = (DJField) x;
              addSource(f,"Remove this field");
              addSource("Create a new file for class "+f.getName()+
                                          " in package "+s_package);
            }
          });

        s_codeUses.map(new Monad() {
            public void f(Object x)
            {
              UsesField u = (UsesField) x;
              addSource(u,"Convert use to instanceof expression");
            }
          });

      }

      public String toString()
      {
        return "specialize "+s_class+" by type code "+
                s_code+" and values "+s_values;
      }

      // command matching

      protected Class[] getArgTypes()
      {
        return new Class[]
          {
            DJField.class, List.class
          };
      }

      protected RefVerb getVerb()
      {
        return RefVerb.SPECIALIZE;
      }

      protected Refactoring makeInstance(Object[] args) throws Exception
      {
        return new SpecTypeCode((DJField) args[0], new EltSet((List) args[1]));
      }

    }// SpecTypeCode
```

Appendix D

The Bicycle Redesign Scenario

This appendix contains the source code for the Bicycle class, and refactoring script for the redesign scenario decomposing it.

D.1 Source Code

```
package edu.mit.infoarch.drjones.bicycle;

public class Bicycle
{

  public static final int MOUNTAIN = 1;
  public static final int ROAD = 2;
  public static final int HYBRID = 3;

  private final int frameType;

  private boolean isCustom;

  private int suspensionType;

  private int wheelSize;

  private int frontGears;

  private int rearGears;

  public Bicycle(int aFrameType)
  {
    frameType = aFrameType;
  }

  public boolean isCustom()
  {
    return isCustom;
  }

  public void setCustom(boolean custom)
  {
    isCustom = custom;
```

```
}

public int getWheelSize()
{
  return wheelSize;
}

public void setWheelSize(int wheelSize)
{
  this.wheelSize = wheelSize;
}

public boolean checkSpecs()
{
  switch (frameType) {
  case MOUNTAIN:
    return (getWheelSize() < 18);
  case ROAD:
    return (getWheelSize() > 20);
  case HYBRID:
    return true;
  default:
    // never happens
    return false;
  }
}

public float getPrice()
{
  switch (frameType) {
  case MOUNTAIN:
    return 500.0f;
  case HYBRID:
    return 300.0f;
  case ROAD:
    return 800.0f;
  default:
    // never happens;
    return -1.0f;
  }
}

public int getFrameType()
{
  return frameType;
}

public int getNumGears()
{
  return frontGears*rearGears;
}

public int getSuspension()
{
  return suspensionType;
}

public void setSuspension(int suspensionType)
{
```

```
    this.suspensionType = suspensionType;
  }

} // Bicycle
```

D.2 Refactoring Script

This is the refactoring script for the Bicycle redesign scenario. Each line in the script that does not begin with "#" or "bookmark" is a single refactoring command, with arguments specified in the syntax

$$\mathit{package}!\mathit{class}.\mathit{member}\ \ [: \mathit{keyword} \ldots]$$

Omitted parts of an argument inherit defaults from the last fully-qualified argument. A keyword is an additional flag passed to the refactoring; in this script the `:remove` keyword is used to tell DR. JONES to remove a method after it has been pushed down. (This is the same as performing a Remove method immediately after the Push Down.)

Lines beginning with "#" are comments, and lines beginning with "bookmark" rename the current node in the design space.

```
# decompose Bicycle by type code

specialize edu.mit.infoarch.drjones.bicycle!Bicycle.frameType \
           .MOUNTAIN .HYBRID .ROAD

bookmark SpecializedByFrameType

# push down suspension-related methods

specialize !Bicycle.getSuspension()I !Mountain !Hybrid :remove
specialize !Bicycle.setSuspension(I)V !Mountain !Hybrid :remove

bookmark PushedDownSuspension

# create subclasses for customizable bikes
specialize !Mountain "CustomMountain"
specialize !Hybrid "CustomHybrid"
specialize !Road "CustomRoad"

bookmark SpecializedCustom

# push down setters

# setCustom(Z)V
specialize !Bicycle.setCustom(Z)V !Mountain !Hybrid !Road :remove
specialize !Mountain.setCustom(Z)V !CustomMountain :remove
specialize !Hybrid.setCustom(Z)V !CustomHybrid :remove
specialize !Road.setCustom(Z)V !CustomRoad :remove

bookmark PushedDownSetCustom

# setFrontGears(I)V
specialize !Bicycle.setFrontGears(I)V !Mountain !Hybrid !Road :remove
specialize !Mountain.setFrontGears(I)V !CustomMountain :remove
specialize !Hybrid.setFrontGears(I)V !CustomHybrid :remove
specialize !Road.setFrontGears(I)V !CustomRoad :remove

bookmark PushedDownSetFrontGears
```

```
# setRearGears(I)V
specialize !Bicycle.setRearGears(I)V !Mountain !Hybrid !Road :remove
specialize !Mountain.setRearGears(I)V !CustomMountain :remove
specialize !Hybrid.setRearGears(I)V !CustomHybrid :remove
specialize !Road.setRearGears(I)V !CustomRoad :remove

bookmark PushedDownSetRearGears

specialize !Mountain.setSuspension(I)V !CustomMountain :remove
specialize !Hybrid.setSuspension(I)V !CustomHybrid  :remove

bookmark PushedDownSetSuspension

# setWheelSize(I)V
specialize !Bicycle.setWheelSize(I)V !Mountain !Hybrid !Road  :remove
specialize !Mountain.setWheelSize(I)V !CustomMountain :remove
specialize !Hybrid.setWheelSize(I)V !CustomHybrid :remove
specialize !Road.setWheelSize(I)V !CustomRoad :remove

bookmark PushedDownSetWheelSize

# generalize subclasses for customizable bikes
generalize "Customizable" !CustomMountain !CustomHybrid !CustomRoad

# we're done with alternative 1

bookmark SubclassingDesign

# navigate back to original design and start alternative 2

# encapsulate frameType

encapsulate edu.mit.infoarch.drjones.bicycle!Bicycle.frameType \
            .HYBRID .MOUNTAIN .ROAD

bookmark EncapsulatedByFrameType

# create subclass for customizable bikes
specialize !Bicycle "CustomBicycle"

bookmark SpecializedBicycle

# push down setters
specialize !Bicycle.setCustom(Z)V !CustomBicycle :remove
bookmark PushedDownCustom

specialize !Bicycle.setFrontGears(I)V !CustomBicycle :remove
bookmark PushedDownFrontGears

specialize !Bicycle.setRearGears(I)V !CustomBicycle :remove
bookmark PushedDownRearGears

specialize !Bicycle.setSuspension(I)V !CustomBicycle :remove
bookmark PushedDownSuspension

specialize !Bicycle.setWheelSize(I)V !CustomBicycle :remove

bookmark DelegationDesign
```

```
# end of script
```

Appendix E

The JUnit Design Scenario

This appendix contains the source code and refactoring script for the design development of the JUnit unit testing framework (Gamma and Beck, 2000).

E.1 Source Code

```
package junit.framework;

public abstract class TestCase extends Assert
{

  /** Return a new TestCase. */
  public TestCase() {
  }

  /**
   * Returns a string representation of the test case
   */
  public String toString() {
    return "(" + getClass().getName() + ")";
  }

  /**
   * Run the test case.
   *
   */
  public abstract void run();

}// TestCase
```

E.2 Refactoring Script

For an explanation of the syntax of this script, consult the script for the Bicycle redesign scenario (Section D.2).

```
# template method pattern

decompose junit.framework!TestCase.run()V "runTest()V"
hide !TestCase.runTest()V
decompose !TestCase.run()V "setUp()V"
hide !TestCase.setUp()V
decompose !TestCase.run()V "tearDown()V"
hide !TestCase.tearDown()V

bookmark TemplateMethod

# collecting parameter pattern
create "TestResult" junit.framework

# failures
create "failures" !TestResult !AssertionFailedError "*"

# errors
create "errors" !TestResult java.lang!Throwable "*"

# add result parameter
create "result" junit.framework!TestCase.run()V !TestResult

bookmark CollectingParameter

# pluggable selector pattern

create "name" !TestCase java.lang!String "!"

bookmark PluggableSelector

# composite pattern

# create composite class TestSuite

create "TestSuite" junit.framework

create "run" !TestSuite "V"
create "result" !TestSuite.run()V !TestResult

create "name" !TestSuite java.lang!String "!"

# generalize common interface Test

generalize "Test" junit.framework!TestCase !TestSuite

# add addTest(Test) method to composite

create "tests" !TestSuite !Test "*"

bookmark Composite

# adapter pattern
```

```
specialize !TestCase "MyTests"
create "myFirstTest" !MyTests "V"
create "mySecondTest" !MyTests "V"
create "myThirdTest" !MyTests "V"
specialize !TestCase.runTest()V !MyTests

specialize !MyTests "AnonymousTest"
specialize !MyTests.runTest()V !AnonymousTest

bookmark Adapter

# add observer pattern

generalize "TestListener" !TestResult
generalize !TestListener !TestResult.addError(Ljava/lang/Throwable;)V
generalize !TestListener \
           !TestResult.addFailure(Ljunit/framework/AssertionFailedError;)V

create "listeners" !TestResult !TestListener "*"

bookmark Observer

# JUnit variations;
# navigate back to Composite and branch from there

# StressTest

specialize !TestCase "StressTest"
specialize !TestCase.run(Ljunit/framework/TestResult;)V !StressTest
create "test" !StressTest !Test "!"
hide !StressTest.setTest(Ljunit/framework/Test;)V
create "threads" !StressTest java.lang!Thread "*"
create "maxThreads" junit.framework!StressTest java.lang!Integer "!"
hide junit.framework!StressTest.setMaxThreads(Ljava/lang/Integer;)V

bookmark StressTest

# Pattern tests

specialize junit.framework!TestCase "SingletonTest"
specialize !TestCase.setUp()V !SingletonTest
specialize !TestCase.runTest()V !SingletonTest
specialize !TestCase.tearDown()V !SingletonTest

specialize junit.framework!TestCase "ObserverTest"
specialize !TestCase.setUp()V !ObserverTest
specialize !TestCase.runTest()V !ObserverTest
specialize !TestCase.tearDown()V !ObserverTest

bookmark PatternTests

# RegressionTest

specialize junit.framework!TestCase "RegressionTest"
specialize !TestCase.run(Ljunit/framework/TestResult;)V !RegressionTest
create "oldTest" !RegressionTest !Test "!"
create "oldResult" !RegressionTest !TestResult "!"
hide !RegressionTest.setOldResult(Ljunit/framework/TestResult;)V
```

```
create "newTest" !RegressionTest !Test "!"
create "newResult" !RegressionTest !TestResult "!"
hide !RegressionTest.setNewResult(Ljunit/framework/TestResult;)V

bookmark RegressionTest

# end of script
```

Bibliography

Balzer, R. (August 1985). Automated enhancement of knowledge representations. In *Proceedings of IJCAI 1985*, pages 203–207.

Banerjee, J. *et al.* (December 1987). Semantics and implementation of schema evolution in object-oriented databases. *SIGMOD Record*, **16**(3):311–322. Special Issue on SIGMOD '87.

Beck, K. (2000). *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Reading, Massachusetts, USA.

Beck, K. and Fowler, M. (1999). Bad smells in code. In *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Object Technology Series, chapter 3. Addison-Wesley, Reading, MA, USA.

Beck, K. and Gamma, E. (2000). Test infected: Programmers love writing tests. `http://junit.sourceforge.net/doc/testinfected/testing.htm`.

Belady, L. A. and Lehman, M. M. (1985). Programming system dynamics for the metadynamics of systems in maintenance and growth. In *Program Evolution: Processes of Software Change*, Lehman and Belady, editors, number 27 in APIC Studies in Data Processing, chapter 5. Academic Press, London. Originally IBM Research Report RC3546, 1971.

Booch, G. *et al.* (2001). Universal Modeling Language specification version 1.3. Published by the Object Modeling Group.

Bowdidge, R. W. and Griswold, W. G. (April 1998). Supporting the restructuring of data abstractions through manipulation of a program visualization. *ACM Transactions on Software Engineering and Methodology*, **7**(2):109–157.

Bracha, G. *et al.* (April 2001). Adding generics to the java programming language: Participant draft specification. Sun Java Community Process JSR14. `http://jcp.org/aboutJava/communityprocess/review/jsr014/`.

Brooks, F. P. (1995). No silver bullet. In *The Mythical Man-Month: Essays On Software Engineering*, anniversary edition edition. Addison-Wesley, Reading, MA, USA. Orig. pub. Proc. IFIP Tenth World Computing Conference 1986.

Casais, E. (1994). Automatic reorganization of object-oriented hierarchies: A case study. *Object-Oriented Systems*, **1**(2):95–115.

Cinne'ide, M. O. (October 2000). *Automated Application of Design Patterns: A Refactoring Approach*. Ph.D. thesis, Department of Computer Science, Trinity College, Dublin, Ireland.

Cinne'ide, M. O. and Nixon, P. (August 1999). A methodology for the automated introduction of design patterns. In *Proceedings of the International Conference on Software Maintenance (ICSM 1999)*. Oxford, England.

Cohen, P. R. *et al.* (1997). QuickSet: Multimodal interaction for distributed applications. In *Proceedings of Fifth Interational Multimedia Conference*, pages 31–40.

Conklin, E. J. and Burgess-Yakemovic, K. (1996). A process-oriented approach to design rationale. In (Moran and Carroll, 1996), chapter 14, pages 393–427.

Egyed, A. (September 2000). Semantic abstraction rules for class diagrams. In *Proceedings of Fifteenth IEEE International Conference on Automated Software Engineering*, pages 301–304. Grenoble, France.

Eichelberger, H. (2002). Aesthetics of class diagrams. In *Proceedings of the First International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2002)*.

Eick, S. G., Graves, T. L., Karr, A. F., Mockus, A., and Schuster, P. (April 2002). Visualizing software changes. *IEEE Transactions on Software Engineering*, **28**(4).

Ernst, M. D., Cockrell, J., Griswold, W. G., and Notkin, D. (February 2001). Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, **27**(2):99–123.

Fischer, G., Lemke, A. C., McCall, R., and Morch, A. I. (1996). Making argumentation serve design. In (Moran and Carroll, 1996), chapter 9, pages 267–293.

Florjin, G. (2002). RevJava – design critiques and architectural conformance checking for Java software. Technical report, Software Engineering Research Centre, The Netherlands.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Object Technology Series. Addison-Wesley, Reading, MA, USA. With contributions by Kent Beck, John Brandt, William Opdyke, and Don Roberts.

Furnas, G. (1986). Generalized fisheye views. In *Proceedings of CHI '86: ACM Conference on Human Factors in Software*, pages 16–23. Association for Computing Machinery.

Gamma, E. and Beck, K. (2000). Junit a cook's tour. `http://junit.sourceforge.net/doc/cookstour/cookstour.htm`.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, Reading, Massachusetts.

Gansner, E. R. and North, S. C. (November 2000). An open graph visualization system and its applications to software engineering. *Software – Practice and Experience*, **30**(11):1203–1233.

Griffin, C. (December 2002). Using EMF. `http://www.eclipse.org/emf/`.

Griswold, W. G. (1991). *Program Restructuring as an Aid to Software Maintenance*. Ph.D. thesis, University of Washington Department of Computer Science, Seattle, WA, USA.

Griswold, W. G. and Notkin, D. (July 1993). Automated assistance for program restructuring. *ACM Transactions on Software Engineering and Methodology*, **2**(3):228–269.

Griswold, W. G., Yuan, J. J., and Kato, Y. (May 2001). Exploiting the map metaphor in a tool for software evolution. In *Proceedings of the 2001 International Conference on Software Engineering*. Toronto, Ontario, Canada.

Griswold, W. G. *et al.* (1998). Tool support for planning the restructuring of data abstractions in large systems. *IEEE Transactions on Software Engineering*, **24**(7):534–558.

Hammond, T. and Davis, R. (March 2002). Tahuti: A geometrical sketch recognition system for UML class diagrams. In *Proc. 2002 AAAI Spring Symposium on Sketch Understanding*.

Jackson, D. (February 2000). Alloy: A lightweight object modelling notation. Technical Report 797, MIT Laboratory for Computer Science.

Jerding, D. F., Stasko, J. T., and Ball, T. (May 1996). Visualizing message-passing patterns in object-oriented programs. Technical Report GIT-GVU-96-15, Georgia Institute of Technology.

Johnson, R. E. and Opdyke, W. F. (1993). Refactoring and aggregation. In *Proc. ISOTAS 1993*. Also appears in Object Technology for Advanced Software, Springer Verlag LNCS 742 pp. 264-278.

Joy, B., Steele, G., Gosling, J., and Bracha, G. (1998). *The Java Language Specification*. Second edition. Addison-Wesley.

Kataoka, Y., Ernst, M. D., Griswold, W. G., and Notkin, D. (November 2001). Automated support for program refactoring using invariants. In *Proceedings of the International Conference on Software Maintenance (ICSM 2001)*. Florence, Italy.

Kunz, W. and Rittel, H. (1970). Issues as elements of information systems. University of California at Berkeley Institute of Urban and Regional Development Working Paper No. 131.

Landay, J. A. and Myers, B. A. (1995). Interactive sketching for the early stages of user interface design. In *Proceedings of CHI 1995: Human Factors in Computing Systems*, pages 43–50. Denver, Colorado, USA.

Lange, D. B. and Nakamura, Y. (1995). Interactive visualization of design patterns can help in framework understanding. In *Proceedings of OOPSLA 1995*. Austin, Texas, USA.

Lanza, M. and Ducasse, S. (2001). A categorization of classes based on the visualization of their internal structure: The class blueprint. In *Proceedings of OOPSLA 2001*, pages 300–311. Tampa, Florida, USA.

Lanza, M. and Ducasse, S. (2002). Understanding software evolution using a combination of software visualization and software metrics. In *Langages et Modèles à Objets (LMO 2002)*, pages 135–149.

Li, H., Reinke, C., and Thompson, S. (August 2003). Tool support for refactoring functional programs. In *Proceedings of Haskell 2003*. Uppsala, Sweden.

Maruyama, K. and Shima, K. (1999). Automatic method refactoring using weighted dependence graphs. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 236–245.

von Mayrhauser, A. and Vans, A. M. (August 1995). Program comprehension during software maintenance and evolution. *IEEE Computer*, **28**(8):44–55.

Moore, I. (1995). Automatic inheritance hierarchy restructuring and method refactoring. In *Proceedings of OOPSLA 1996*. San Jose, California, USA.

Moran, T. P. and Carroll, J. M., editors (1996). *Design Rationale: Concepts, Techniques, and Use*. Lawrence Erlbaum.

Müller, H., Tilley, S., Orgun, M., Corrie, B., and Madhavji, N. (December 1992). A reverse engineering environment based on spatial and visual software interconnection models. *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments (SIGSOFT 1992)*, **17**(5):88–98. ACM Software Engineering Notes.

Murphy, G. C., Notkin, D., and Sullivan, K. J. (April 2001). Software reflexion models: Bridging the gap between design and implementation. *IEEE Transactions on Software Engineering*, **27**(4):364–380.

Oltmans, M. (2000). *Understanding Naturally Conveyed Explanations of Device Behavior*. Master's thesis, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, USA.

Opdyke, W. (1992). *Refactoring Object-Oriented Frameworks*. Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, USA.

Oviatt, S. L. *et al.* (2000). Designing the user interface for multimodal speech and gesture applications: State-of-the-art systems and research directions. *Human Computer Interaction*, **15**(4):263–322.

Parnas, D. L. (1979). Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, **5**(2).

Reubenstein, H. B. (June 1990). Automated acquisition of evolving informal descriptions. Technical Report AI-TR-1205, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, USA.

Rich, C. and Waters, R. C. (November 1987). The Programmer's Apprentice project: A research overview. Technical Report AI Memo 1004, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, USA.

Rich, C. and Waters, R. C. (January 1989). Intelligent assistance for program recognition, design, and debugging. Technical Report MIT AI Memo 1100, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, USA.

Rilling, J., Seffah, A., and Bouthlier, C. (2002). The CONCEPT project – applying source code analysis to reduce information complexity of static and dynamic visualization techniques. In *Proceedings of the First International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2002)*.

Roberts, D. (1999). *Practical Analysis for Refactoring*. Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, USA.

Roberts, D., Brandt, J., and Johnson, R. (1997). A refactoring tool for Smalltalk. *Theory and Practice of Object Systems (TAPOS)*, **3**(4):253–263.

Robillard, M. P. and Murphy, G. C. (May 2002). Concern graphs: Finding and describing concerns using structural program dependencies. In *Proceedings of the 2002 International Conference on Software Engineering*. Orlando, Florida, USA.

Schauer, R. and Keller, R. K. (June 1998). Pattern visualization for software comprehension. In *Proceedings of the Sixth International Workshop on Program Comprehension (IWPC 1998)*. Ischia, Italy.

Shrobe, H. (April 1979). Dependency directed reasoning for complex program understanding. Technical Report AI-TR-503, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, USA.

Storey, M.-A. D., Fracchia, F. D., and Müller, H. A. (May 1997a). Cognitive design elements to support the construction of a mental model during software visualization. In *Proceedings of the 5th International Workshop on Program Comprehension*, pages 17–28. Dearborn, Michigan.

Storey, M.-A. D., Wong, K., and Müller, H. A. (May 1997b). Rigi: A visualization environment for reverse engineering. In *Proceedings of the 1997 International Conference on Software Engineering*. Boston, Massachusetts, USA.

Systä, T. (June 1999). Dynamic reverse engineering of Java software. In *Proc. ECOOP Workshop on Experiences in Object-Oriented Re-Engineering*.

Systä, T. (November 2000). Understanding the behavior of Java programs. In *Proceedings of Seventh Working Conference on Reverse Engineering*. Brisbane, Austrailia.

Tichelaar, S., Ducasse, S., Demeyer, S., and Nierstrasz, O. (2001). A meta-model for language-independent refactoring. In *Proceedings of International Symposium on Principles of Software Evolution (ISPSE 2001)*, pages 154–164.

Tokuda, L. and Batory, D. (2001). Evolving object-oriented designs with refactorings. *Automated Software Engineering*, **8**:89–120.

Waingold, A. and Jackson, D. (1999). Lightweight extraction of object models from bytecode. In *Proceedings of the 1999 International Conference on Software Engineering*.

Walker, R. J., Murphy, G. C., Steinbok, J., and Robillard, M. P. (November 2000). Efficient mapping of software system traces to architectural views. In *Proceedings of CASCON 2000*. Mississauga, Canada.

Walker, R. J., Murphy, G. C., *et al.* (1998). Visualizing dynamic software system information through high-level models. In *Proceedings of OOPSLA 1998*. Vancouver, British Columbia, Canada.

Wills, L. M. (July 1992). Automated program recognition by graph parsing. Technical Report AI-TR-1358, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, USA.

Zannier, C. and Maurer, F. (2002). Tool support for complex refactoring to design patterns. In *Proceedings of OOPSLA 2002*. Seattle, Washington, USA.